University of Alberta

Code and Data Outlining

by

Peng Zhao ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta

Fall 2005

To Fang,

without a doubt

Abstract

In this dissertation we investigate compiler techniques to address the performance problems caused by heterogeneous execution frequency of code in the same function and heterogeneous access pattern of fields in the same data structure. These heterogeneous characteristics are bad for performance. On one hand, it is frequent that instructions in the same function have very different execution frequencies. There is often infrequently referenced *cold* code, such as exception handlers, intertwined in frequently invoked *hot* functions. Cold code in hot functions not only degrades instruction cache efficiency but also makes host functions too large to be inlined. On the other hand, programmers organize their data layout in a semantically meaningful way that often does not match the runtime access pattern well. This data organization causes inefficient data cache utilization.

We use compiler outlining techniques to address these performance problems that are difficult to handle by programmers. For programs with heterogeneous execution frequency, we use function outlining to split cold code out of the host function. Function outlining makes the host function smaller and more amenable for inlining optimization because the compiler is then able to do partial inlining, *i.e.* inline only the hot parts of a callee. To address the heterogeneous data pattern issue, we use data outlining or reshaping, which splits large data structures into smaller ones, to improve the efficiency of data cache.

We describe in detail the necessary analysis and transformations needed to preserve correct program behavior in code and data outlining. In both function outlining and data outlining, we conduct a study of possible strategies. Our study shows that, although function outlining can be used to reduce function sizes (by up to 97%) and partial inlining improves performance by up to 5.75%, partial inlining has very limited effect on enabling more aggressive inlining for SPEC2000 benchmarks. The major benefits of partial inlining are actually the benefits of function outlining, which become more pronouncing when inlining is enabled. We also found that data reshaping could improve performance dramatically: one of the benchmarks studied achieves 2.1 times speedup with proper reshaping strategy. Detailed analysis explains these performance results.

Acknowledgements

I want to heartedly thank everybody who generously offered me help during my quest of Ph.D degree.

First of all, I am indebted to my advisor, Dr. José Nelson Amaral, for his numerous valuable discussions and suggestions. It is he who introduced me to the field of compiler research. I also appreciate his patience and support throughout my entire project.

Thanks also go to the other members of my Ph.D. exam committee members, Dr. Bruce Cockburn, Dr. David Padua, Dr. Jonathan Schaeffer, Dr. Lorna Stewart, and Dr. Duane Szafron. I thank them for agreeing to serve in my candidacy exam committee and for their time and energy spent on reviewing my proposal and thesis. Especially, their discussions during my candidate exam made me start to think about the possibility of data outlining, which is the second part of this thesis.

I thank Sun C. Chan and Shin-ming Liu for their insightful discussions on function outlining. I also thank Dr. Yaoqing Gao, Shimin Cui and Raúl Silvera for their discussions and cooperation on my data-outlining work at IBM. I got much help from the mailing lists for Open64, ORC, Pfmon, and Pro64 when I picked up my experimentation platforms. Here I specially thank Kaiyu Chen, Buqi Cheng, Jim Dehnert, Stephane Eranian, Lixia Liu, Michael Murphy, Chandrasekhar Murthy, Chengyong Wu, Shuxin Yang, Qingyu Zhao, and Shukang Zhou. My project would have been much more difficult without their kind help.

Last, but not least, this dissertation would not be possible without the help from my family. I thank my parents, Weigui Zhao and Yueqing Zhao, for their moral support throughout this five-year journey. My wife, Fang Liu, has demonstrated her unconditional love by numerous encouragements when I felt frustrated and by sacrificing the living standards that she has long deserved to be with me.

Financial support for this dissertation was provided by the Natural Sciences and Engineering Research Council (NSERC) of Canada. I also thank Intel Corp. and IBM Corp. for the internship opportunities that allowed me to expand my work on function outlining and data outlining, respectively.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Research Goal

Over the past four decades, we have been witnessing the ever-increasing speed of modern microprocessors due to advances in semiconductor fabrication and architectural innovation. On the other hand, people are working hard to improve programmer productivity by adopting software engineering techniques that emphasize modularity, code reuse, and maintainability. There is a gap between performance engineering and software engineering. Software productivity is often emphasized and the programs produced with advanced software engineering techniques are often suboptimal in performance because they cannot fully utilize the resources or architectural features of the underlying hardwares. This gap is still widening. A good compiler should bridge this gap by transforming high-level applications into hardware-friendly binaries.

In this dissertation, we deal with two specific problems arising from the semantic gap between software engineering and performance engineering. The first problem stems from the heterogeneous frequency[1] of code in a single function. When writing a program, programmers tend to place semantically-related code together in a function, even though the execution frequencies of these codes might differ significantly. A good example is error and exception handling code. As a consequence, there are often many cold (*i.e.* infrequently-executed) code segments in a hot function. Putting code segments with heterogeneous execution frequencies together deteriorates instruction cache effi-

[1] In this thesis, we call the number of occurences of executing a piece of code the execution frequency of the code. Similarly, the number of times a piece of data is accessed is its access frequency. Heterogeneous frequency is a conceptual term. When two frequencies are very different (*e.g.* 0 versus 100000), we say they are heterogeneous.

ciency, interferes with a compiler's goal of focusing its resources and time on frequently-executed code, and, more importantly, makes host functions too large to be inlined.

The second problem studied in this thesis is the performance penalty due to a semantics-oriented data layout. In high-level programming languages, programmers use aggregate data structures, such as `structs` and `classes`, to organize their data. All the features of a given object are put together in the same aggregate data structure, without consideration of runtime access patterns. This data layout also prevents better performance for several reasons. First, the frequency of access to fields in the same data type may vary significantly with *hot fields* accessed very frequently and *cold fields* seldom referenced. Placing fields with very different access frequencies together in memory hurts performance because the cold fields pollute the data cache and waste memory bandwidth. Second, the runtime data access pattern might not be consistent with the access frequency distribution. In other words, hot fields are not necessarily accessed together. This means that the temporal locality of certain hot fields is degraded by other hot fields.

During preliminary investigation toward this thesis, we spotted the above-mentioned problems in some popular benchmarks. This motivated us to investigate the performance potential of compiler techniques to address these problems. Since the performance setback of heterogeneous characteristics in both instruction and data were unclear at the start of our study, we decided to study both.

Generally, the unifying technique of this thesis, outlining, consists of removing a cold part from a hot host function or data structure. To handle the cold code segments in a hot function, we use function outlining to split them out of their host functions and generate new functions to hold these split code segments. The original code segments in the host functions are replaced by function calls to the new functions. Function outlining separates hot code from cold code and makes the original hot function smaller and therefore more amenable to inlining optimization. Because only the hot portions in a function are inlined after function outlining, we call it *partial inlining.* To make the data layout more friendly to the underlying hardware, we split an object into two or more smaller objects. Each smaller object holds fields that have high access affinity. Accordingly, we split arrays that contain aggregate data structures into multiple smaller arrays. The result of the data outlining is a data layout that has a smaller memory footprint, has better locality and is more amenable to hardware data prefetching.

Important research questions include:

- Would function outlining followed by partial inlining yield performance improvements in industry-standard benchmarks? To achieve partial inlining, we need region identification algorithms to find the cold code segments in a hot function. Sometimes the cold code in a function is not well organized. Therefore, we need some transformations to reorganize the codes so that they are easier to split. Regions that are split out of the host function might cause extra runtime function calls. We could suffer serious performance degradation if hot codes are split out of a host function. To gain performance by partial inlining, the performance degradation of function outlining itself must be kept strictly under control. Hence, function outlining must achieve a balance between reduced host function sizes and potential increase in runtime function calls. Finally, to make function outlining safe, we also need to preserve correct program behavior when splitting code out of a function.

- Can data outlining be made safe to be integrated into a production-level compiler? The transformed program must retain its original semantics. Because of the prevailing usage of pointers and type casting in the C/C++ programming languages, transforming data accesses is not a trivial task. If two data accesses are aliased with each other, transforming one of them means that we must also update the other one. Also, data outlining should be avoided if a memory location is referenced through multiple views; it is dangerous to do transformation without a consistent starting point. If the safety problem is solved, does data outlining results in performance improvements?

1.2 Contributions

The primary contributions in this dissertation include:

- By carefully tuning the inlining framework in the Open Research Compiler (ORC), we found that large function bodies are among the major impediments to beneficial inlining. This suggests an opportunity for function outlining and partial inlining.

- In our function outlining work, we propose a region formation algorithm based on an abstract syntax tree. Our region formation algorithm efficiently exploits high-level control flow structures, and their associated feedback information, to identify candidate regions for outlining. We formulate the *Optimal Outlining Problem* and argue that it is NP-hard. Then we devise an effective heuristic to analyze the benefits of outlining a region. This heuristic decision weighs the benefit of reducing the host function size against the execution frequency of the extra function calls introduced. We describe how to patch the control flow and the data flow to preserve the program semantics in outlining. Because outlining is an early code transformation, it may negatively impact existing downstream optimizations. Our experiments show that complex alias relationships created by outlining parameters have a major impact on downstream optimizations and may result in the introduction of substantial memory spills. To handle this problem, we propose a novel technique, called *alias agent*, to disambiguate parameters created to pass references to outlined functions. Finally, we study two orthogonal function splitting strategies: (1) *collective* versus *independent* splitting; and (2) splitting with versus without alias agent. This study shows that selecting the correct strategy is crucial. Independent splitting with alias agent reduces function sizes significantly while minimizing the performance penalty of outlining. Based on our function outlining work, we report the performance results of partial inlining.

- We build *Forma*, a practical data outlining (or data reshaping) framework that can be used to automatically analyze and transform C/C++ programs. *Forma* consists of a data shape analysis, including both alias analysis and data type analysis, structure partition planing and array reshaping transformations. *Forma* has been integrated into the IBM XL C/C++ V7.0 compiler. We also conduct an empirical study of two orthogonal reshaping decisions: frequency-based object partition × affinity-based object partition × maximum object partition; and address-arithmetic-based × pointer-based array splitting. Some important, but subtle, insights on data reshaping are exposed by a thorough analysis and empirical study. We found that data reshaping could improve performance significantly. Experimental results also suggest that the combination of address-arithmetic-based array splitting and the seemingly naive maximal object splitting achieves the best or close-to-best performance on the studied benchmarks.

1.3 Dissertation Organization

Both function outlining and data outlining are inter-procedural optimizations that require runtime profiling information to estimate optimization benefits. Therefore, as a background introduction, Chapter 2 describes important concepts that are directly relevant to this thesis. We first introduce inter-procedural optimization (IPO) and profiling-guided optimization. Then we use the Open Research Compiler (ORC) as an example to demonstrate how a compiler's behavior changes when IPO is involved. Chapter 3 investigates the inlining trade-offs and identifies the partial inlining opportunities in the SPEC INT2000 benchmarks. Chapter 4 proposes function outlining and discusses its design and implementation. We also report our results of partial inlining in Chapter 4. Chapter 5 handles the data layout problem and presents *Forma*, a data outlining framework that is safe and automatic. We conclude this dissertation in Chapter 6 by summarizing our contributions and by presenting some extension opportunities.

Chapter 2

Background

This section introduces the background for this dissertation. The research in this dissertation consists of two inter-procedural optimizations (IPO): function outlining and data outlining. Both of them require inter-procedural analysis (IPA) and runtime feedback information. Therefore, in this section we discuss inter-procedural optimizations and profiling-guided compiler optimizations. We use the Open Research Compiler (ORC) [1], one of our research platforms, as an example to demonstrate how a compiler's behavior changes when IPO is involved.

2.1 Inter-Procedural Optimization (IPO)

The placement of related program segments into separate procedures hides useful information, limits the scope of compiler analysis, and prevents the aggressive application of compiler optimizations. Classic optimizations, such as common sub-expression elimination, constant propagation, alias analysis, code scheduling, and register allocation, cannot be easily applied across procedure boundaries. In an intra-procedure compilation framework, the boundaries of procedures are barriers that prevent compiler optimization. In the example shown in Figure 2.1, assume that there is only one place in the entire program that calls ***bar***. A compiler that has no cross-boundary information about ***main*** and ***bar*** has to conservatively assume that the values of the parameters passed to ***bar*** are only known at runtime.

However the second parameter to the invocation of ***bar***, variable ***a*** in ***main*** is a constant that can be propagated from ***main*** to ***bar***, as shown in Figure 2.2

After the constant propagation, the second parameter to ***bar*** is not needed anymore and can be eliminated. Also, the definition of variable ***a*** in main is

```
void
main()
{
    int a, i, sum;
    sum = 0;
    a = 10;
    for(i=0; i<1000000; i++) {
        sum += bar(i, a) /1000;
    }
    printf("%d\n", sum);
}
```

```
int
bar(int Li, int La )
{
    int j, result=0;
    for(j=0; j<Li; j++)
        result += La *j;
    return result;
}
```

Figure 2.1: The procedure barrier against compiler optimization

```
void
main()
{
    int a, i, sum;
    sum = 0;
    a = 10;
    for(i=0; i<1000000; i++) {
        sum += bar(i, 10) /100;
    }
    printf("%d\n", sum);
}
```

```
int
bar(int Li, int La )
{
    int j, result=0;
    for(j=0; j<Li; j++)
        result += 10 *j;
    return result;
}
```

Figure 2.2: Inter-procedural constant propagation on variable a

no longer used, and therefore can be eliminated by dead code elimination. These transformations produce the code shown in Figure 2.3. To find these optimization opportunities, a compiler must analyze the data and control flow both in the caller and in the callee.

```
void
main()
{
    int a, i, sum;
    sum = 0;

    for(i=0; i<1000000; i++) {
        sum += bar'(i) /100;
    }
    printf("%d\n", sum);
}
```

```
int
bar'(int Li )
{
    int j, result=0;
    for(j=0; j<Li; j++)
        result += 10 *j;
    return result;
}
```

Figure 2.3: Inter-procedural constant propagation enables function specialization and dead store elimination

Inter-Procedural Optimization (IPO), also called *cross-module optimization* or *whole-program optimization*, improves the performance of programs by exploring optimization opportunities across procedure boundaries. By taking advantage of these opportunities, IPO eliminates the performance penalty associated with small program units, and thus enables programmers to take advantage of software modularity. IPO has been implemented in most modern industry-strength compilers and has proved to be a very effective optimization technique. For instance, in the HP-UX 10.20 compiler, Ayers *et al.* demonstrated a performance improvement of 32% for the SPECint95 benchmark on a PA8000-based workstation [7]. Later, they reported that IPO can speed up independent software vendor's applications, with as many as 5 million lines of source code, by as much as 71% [8].

To be employed in commercial compilers, IPO must be carefully designed. First of all, IPO requires that the compiler be able to analyze and optimize code throughout the application. Usually compilers have access to the whole program only during the linking phase, when all the relocatable object files are read and linked together to create an executable file. Thus, inter-procedural optimizations are often implemented in the linker [65, 66]. We call these inter-procedural optimizations *link-phase IPOs* or *low-level IPOs.* The input to low-level IPOs consists of relocatable instructions. Implementing IPO in the link phase has two major advantages. First, link-phase IPO has access to the whole application because static library code is available in the link-phase but is not easily available in earlier phases. The second advantage is that link-phase optimizations have access to low-level information. Examples of low-level information includes register usage, register availability, function layout, and procedure sizes. Some low-level IPOs, such as inter-procedural register allocation and code placement, can only be performed in the link phase.

However, important IPOs cannot be applied in the link phase because binary instructions in object files have lost required high-level information. A good example is type and aggregate data construct information that is essential for the efficiency of inter-procedural alias analysis. Another important drawback of low-level IPO is that the link phase takes place at the very end of the compilation process, which means that some optimization opportunities originated by IPO have little chance to be explored. Therefore it is also desirable that some inter-procedural optimizations be performed on higher level language representations of the program. We call these inter-procedural optimizations *high-level IPOs.*

A high-level IPO method consists of introducing a fake linker early in the compilation. This fake link phase reads and analyzes all the available source code. However, instead of relocatable binaries, the input and output of the fake linker are both high-level intermediate representations. Hence, after the fake high-level IPO, other classic optimization techniques can be applied efficiently on the high-level intermediate representation.

Many modern compilers, including the MIPSPro compiler [48], the Open Research Compiler, the HP-UX compiler [8], and the IBM XL compiler [24], implement high-level IPO with a fake linking phase.

2.1.1 Tractable IPO

An obstacle to IPO is that, when applied to large applications, it might result in excessive compilation time and excessive memory space requirements for the compiler. Earlier IPO experiences suggest that a naive IPO design cannot be used for large applications. For example, the IPO-enabled compiler in HP-UX9.0 requires, on average, 1.7KB of memory for each line of source code [8], which makes even the compilation of some moderate-sized benchmarks very memory-demanding and time-consuming. The problem is even worse for commercial software with millions of lines of source code. There are several ways to address the IPO scalability issue. First, the data structures in the compiler must be carefully designed. Complex data management techniques might be employed to schedule the loading of information into memory when it is needed and swapping it out to external storage or discard it according to its expected future use. Second, IPO can be selectively applied to the most important units of the program. The 80-20 rule predicts that usually only a very small portion (20%) of the code consumes most of the execution time (80%) [15]. Thus the compiler should focus its optimization efforts on the most executed program units, which consist of only a small portion of the whole application. The selection of the optimization target can be guided by programmer intervention or by profiling information.

2.2 Profiling-guided IPO

Whole program analysis and optimization is often impractical because of its excessive time and memory requirements. Given the large variance in the potential benefit of analyzing and optimizing each call site, IPO must optimize call sites selectively. IPO may increase the size of some procedures through inlining or code duplication and causc longer compilation times. In some cases, IPO may even produce slower programs because of unexpected adverse effects in instruction caches and register usage. Thus, compilers must select call sites that are amenable to inter-procedural optimization. This selection is often informed by profiling information.

Profiling-guided compilation requires more than one compilation and execution of the application before the final executable code is generated. The simplest profiling-guided compilation requires three steps: two rounds of compilation and one round of execution taking place between the compilations. First, the compiler inserts counters to collect run-time statistics. These counters may be inserted either in the original source code or in an intermediate representation of the program. Runtime statistics collected include call site frequencies, the frequency in which each branch is taken or not taken, the number of iterations executed by each loop, and so on. We call this phase the ***instrumentation*** phase. The second phase is the ***instrumented execution*** of the instrumented program with a training data input. The counters inserted in the code collect runtime statistics and save these statistics in a formatted file that can be interpreted by a later compilation. Finally, the runtime statistics gathered during the instrumented execution are used to guide the second compilation. The action of associating the source code, or the intermediate representation, with its respective frequency information is called ***annotation***.

Feedback information is beneficial for optimization when the training data set is representative of typical input data sets. However, profiling-guided compilation comes at a cost. The need for preliminary compilation and for a training execution is inconvenient and time-consuming. Often, the instrumented program needs much more time to compile and to execute. Experiments show that instrumented programs slow down by 30–1,000% when compared with their non-instrumented versions [5, 14, 17, 20, 34]. Thus, efficient profiling has concerned several researchers [6, 11, 13]. An alternative to instrumentation and feedback information is to use an estimation of the execution frequency distribution in an application through static program analysis [12, 19, 29, 40, 90, 97].

Another drawback of profiling-based optimization is that its effectiveness depends on the training input set. The training input must be representative of the actual input at runtime to generate appropriate feedback information. However, research shows that the representativeness is not a big problem in real profiling-based optimizations [32, 90]. Our work is based on profiling information.

2.3 IPO Case Study

In this dissertation, function outlining is conducted using the Open Research Compiler (ORC) and data outlining is conducted using the Toronto Portable Optimizer (TPO). Both compilers handle IPO in a similar fashion. We will use ORC as an example to show how IPO-involved compilation is different from compilation without IPO. ORC is an open-source compiler that is adapted from the SGI MIPSPro compiler. ORC generates executables for Itanium Processor Family (IPF) processors. ORC inherits from the MIPSPro compiler a mature compilation infrastructure, a rich optimization set including IPO, and complete program analysis support. This excellent pedigree makes ORC a very good platform for experimenting with novel research ideas.

2.3.1 ORC without IPO

When IPO is not invoked, ORC works like a traditional compiler as shown in Figure 2.4. ORC transforms source code into executable in two phases. The first phase, shown inside the dotted ellipse in Figure 2.4, is the modular compilation of each source code file to generate a corresponding object code files. The second phase is the linking of the object files into the final executable.

During the first phase, each source code file is first translated by the compiler ***front-end*** (FE) into its intermediate representation called WHIRL (Winning Hierarchical Intermediate Representation Language). The WHIRL object file is then transformed by the compiler ***middle-end*** (ME) and ***back-end***(BE) into a relocatable object file. The WHIRL transformation iterates on all the program units (PUs) in a source code file.

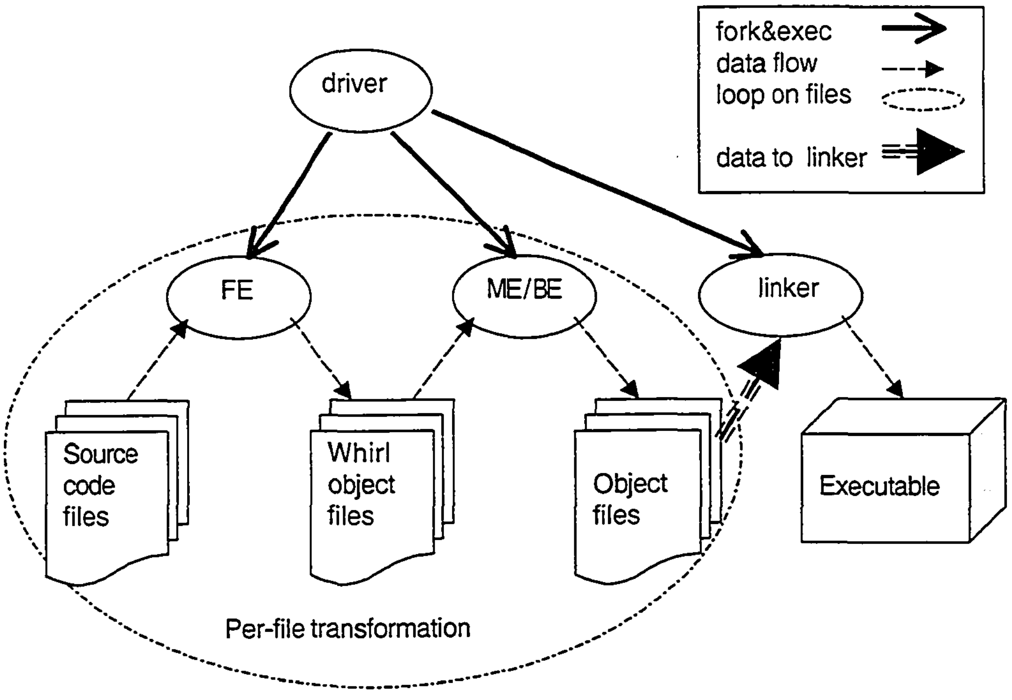


Figure 2.4: ORC compilation without IPO

WHIRL Transformation Levels

The compiler gradually translates the source code from high-level language constructs to low-level machine instructions. As more and more optimization techniques are added into ORC, the robustness and maintainability of the compiler itself becomes an important concern. To address this issue, the WHIRL transformation is further divided into several phases, resulting in several WHIRL levels. The higher the WHIRL level, the closer the WHIRL representation is to the original source code. When the program is transformed from the very high-level WHIRL to the very low-level WHIRL, the hierarchical constructs from the high-level programming language are gradually *lowered* to flat instruction-like constructs.

Optimizations have to be orchestrated carefully so that they take place at the proper WHIRL levels. There might be several reasons for some optimizations to occur at higher levels of WHIRL representation. First, at the higher levels, the compiler can access more accurate information, such as control flow constructs and complex data types, from the original program. For example, some programming language operations are represented by a single high-level WHIRL statement. Thus, it is easy to find redundant operations by testing if the operator and the operands are the same or not. However, once these high-level constructs are lowered to several low-level WHIRL statements, it is more difficult to detect the redundant operations. Moreover, high-level transformations are more efficient and easier to implement because a high-level WHIRL representation typically contains fewer, but more expressive, statements.

On the other hand, some optimizations can only take place on a low-level intermediate representation. For example, register allocation has to be conducted at the end of the compilation because only at that time can the compiler access all the variables, including the original variables in the program and the temporary variables generated by earlier compiler transformations.

Some optimizations, such as dead store elimination and copy propagation, need to be repeated several times throughout the compilation. One reason is that the opportunities for these optimizations appear after other transformations are performed. Moreover, these optimizations, in turn, might enable further optimizations. Another important reason to perform some optimizations multiple times is that, when applied early, they help make the WHIRL representation more concise and more tractable for downstream transformations.

Middle-end and Back-end Optimizations

A large repertoire of code optimizations takes place along with WHIRL transformations. Some of these optimizations are machine-independent while others are very sensitive to the low-level micro-architectural organization. Machine-independent optimizations usually take place before machine-dependent ones. Thus WHIRL transformations are also divided into two phases. The earlier phase is formed by the machine-independent *middle-end* optimizations. Middle-end optimizations include the incremental *lowering* of the intermediate representation of the program, the high-level loop optimizations, and other traditional optimizations. The later phase performs the *back-end* optimizations, which are machine-dependent. Back-end optimizations include register allocation, code scheduling, *etc.*

2.3.2 ORC with IPO

ORC implements a complete IPO facility. Figure 2.5 shows the compilation process in ORC when IPO is invoked. IPO creates opportunities for other middle-end and back-end code optimizations. Therefore IPO must be performed before such optimizations.

Source-code files are fed into the front-end for the generation of intermediate representation in the form of WHIRL-object files. Then Inter-Procedural Lowering (IPL)[1] reads the WHIRL object files and the feedback information, analyzes them one-by-one, and writes the frequencies, estimated size, and estimated number of cycles executed in each procedure into the original WHIRL

[1] IPL is a preparation phase for IPO. Probably ORC uses this term for historical reasons.

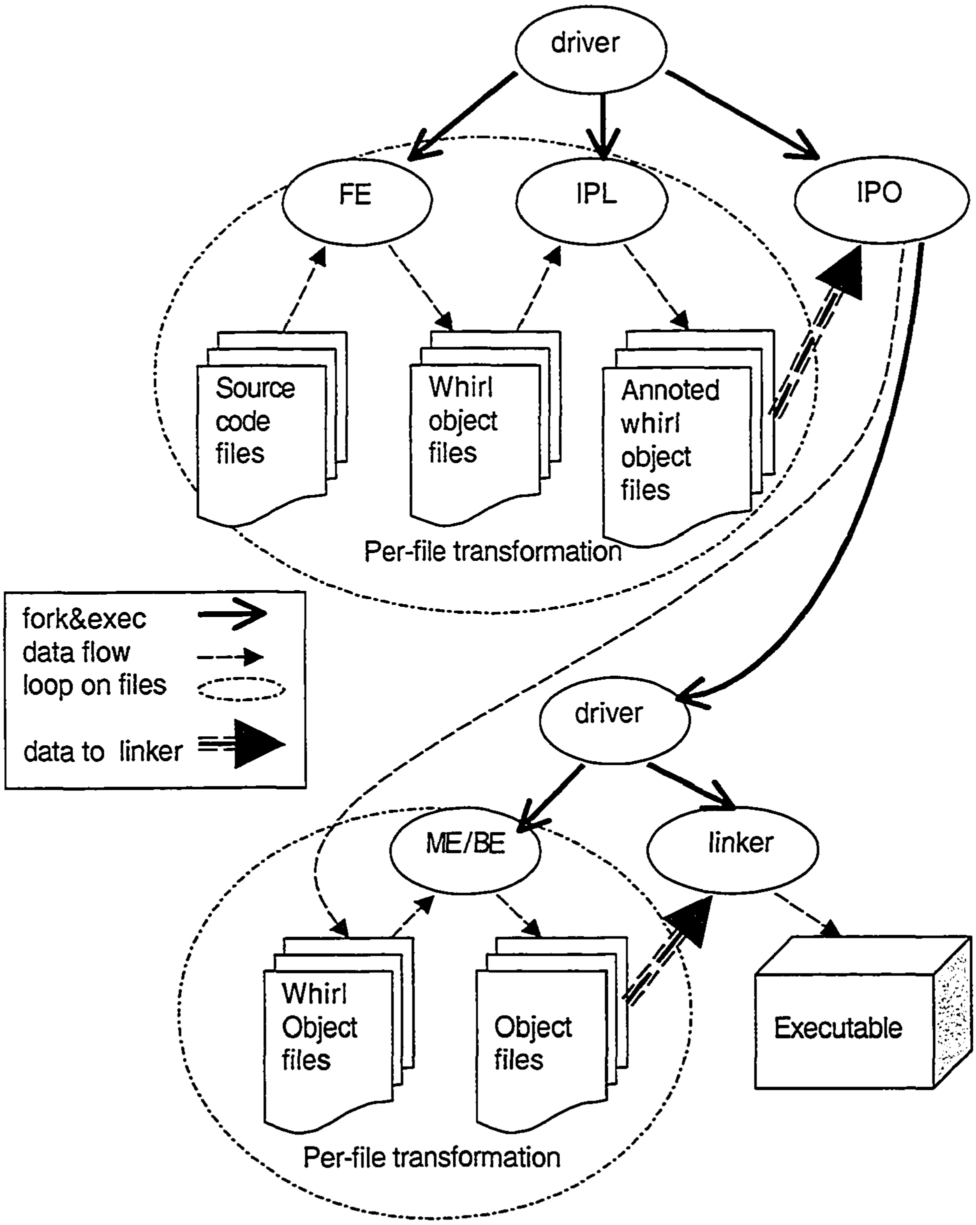


Figure 2.5: IPO-involved ORC compilation

objects into files. IPO takes these annotated intermediate representations as input to inlining and other inter-procedural optimizations. After IPO, the procedures are fed to the traditional optimizer to generate relocatable objects. Finally, the linker is invoked to generate the executable.

From Figure 2.5, we can see that, compared with the front-end, with the back-end, and with the IPL, the operation of IPO is more similar to a linker. The IPO reads all the available WHIRL object files and analyzes them together. This is necessary to enable the compiler to access the whole application instead of a single procedure or source file. By analyzing the information about all the source files, the compiler can perform inlining across the boundaries of different source files. This wider scope for optimization is important for performance because closely-related callers and callees do not always reside in the same source file.

Chapter 3

Inlining Tuning

3.1 Introduction

Function inlining is a very important optimization technique that replaces a function call with the body of the function [7, 22, 25, 26, 27, 30, 47, 59, 87]. One advantage of inlining is that it eliminates the overhead resulting from a function call. The savings are especially pronounced for applications where only a few call sites are responsible for the bulk of the run-time function invocations; inlining those call sites significantly reduces the function invocation overhead. For example, mcf (one of the SPEC2000 benchmarks) contains 34 call sites. Among these call sites, there are 5 that are executed more than 10 million times and 4 call sites that are executed more than 1 million times in a standard SPEC2000 training execution. These 9 call sites account for 99.85% of all the function invocations in mcf. Our experiments show that inlining the 15 most frequent call sites can reduce the running time of mcf by more than 9%.

Inlining also expands the context of static analysis. This wider-scoped analysis creates opportunities for other optimizations. Because the body of the callee is now available at the call site, conservative assumptions that the compiler would previously make about the call site are no longer required.

Another advantage of inlining is the improvement of cache efficiency. From the point of view of the data cache (D-cache), after inlining there is no need to create parameters to pass the caller's variables that are referenced by the callee. Thus, a variable that previously had separate representations in the caller and in the callee can now be reduced to a single memory location or even promoted to a register. This storage consolidation reduces the data access footprint of the application and improves the use of the memory hierarchy. A similar advantage also exists for the instruction cache (I-cache). After inlining, closely related segments of code are placed together, reducing the chances of instruction cache conflicts [72].

However, inlining can have negative effects. One problem with inlining is the growth of the code, also known as *code bloat*. Because a procedure may be called from multiple call sites, it is often not possible to eliminate a procedure after inlining a single call site. Thus, the final executable file must contain several copies of the procedure: the original one and the inlined copies. With the growth of functions because of inlining, the compilation time and the compiler memory space requirement may become intolerable because some of the algorithms used for static analysis have super-linear complexity.

Besides of the compilation time and memory resource cost, inlining might also have the adverse effect of increasing the execution time of the application. After inlining, the register pressure may become a limitation because the caller now contains more code, more variables, and more intermediate values. This additional storage requirement may not fit in the register set available in the machine. Thus, if the register allocator cannot do a good job, inlining may increase the number of register spills resulting in a larger number of load and store instructions executed at runtime.

The above discussion of the benefits and drawbacks of inlining leads to an intuitive criterion to decide which call sites are good candidates for profitable inlining. The value of the benefits of inlining, such as eliminating function-call overhead, enabling more optimization opportunities, and improving cache efficiency, depend on the execution frequency of the call site. The more frequently a call site is invoked, the more promising is the inlining of the site. If the call site is invoked only a couple of hundred times in a long execution, inlining it is unlikely to produce any improvement.

On the other hand, the negative effects of inlining relate to the size of the caller and the size of the callee. Larger functions tend to have worse cache behavior and higher register pressure. Inlining large callees results in more serious code bloat, and, probably, performance degradation due to additional *memory spills*.[1]

Thus, we have two basic guidelines for inlining. First, the call site must be very frequent, and, second, neither the callee nor the caller should be too large. Most of the papers that address inlining take these two factors into consideration in their inlining analysis.

In this chapter we describe our experience in tuning the inlining heuristics for the Open Research Compiler (ORC). The main contributions of this chapter are:

[1] A memory spill occurs when the register allocation algorithm is not able to fit all live values into registers. In this case, some values must be written temporarily (*spilled*) to memory.

- We propose *adaptive inlining* to enable aggressive inlining for small benchmarks. Usually, small benchmarks are amenable to aggressive inlining as shown in Section 3.4. Adaptive inlining becomes conservative for large benchmarks such as gcc because the negative effects of aggressive inlining are often more pronounced in such benchmarks.

- We introduce the concept of *cycle_density* to control code bloat and compilation-time increase.

- Our detailed experimental results show the potential of inlining. We investigate the impediments to beneficial inlining and motivate function outlining and partial inlining.

The rest of this chapter is organized as follows: Section 3.2 describes the existing inlining analysis in ORC and its limitations. Section 3.3 describes our enhancements to the inlining analysis (adaptive inlining and *cycle_density* heuristics) and Section 3.4 studies performance, finds the impediments for inlining frequent call sites, and motivates function outlining and partial inlining. Section 3.5 reviews related work. The findings presented in this chapter were published in [99].

3.2 Overview of ORC Inlining

To control the negative effects of inlining, we should inline selectively. How do we determine whether a call site is suitable for inlining? The performance effect of inlining an edge of the call graph depends on two factors: the execution frequency of the site and the size of the callee. The problem of selecting the most beneficial call sites while satisfying the code bloat constraints can be mapped to the *knapsack* problem, which has been shown to be NP-complete [37, 79]. Thus, heuristics are often used to estimate the gains and the costs of each potential inlining. ORC uses profiling information to calculate the *temperature* of a call site to approximate the potential benefit of inlining an edge $E_i(p,q)$ that represents a call site in a function p that calls a function q.[2]

$$temperature_{E_i(p,q)} = \frac{cycle\_ratio_{E_i(p,q)}}{size\_ratio_q} \tag{3.1}$$

where:

$$cycle\_ratio_{E_i(p,q)} = \frac{freq_{E_i(p,q)}}{freq_q} \times \frac{cycle\_count_q}{Total\_cycle\_count} \tag{3.2}$$

[2] Because function p may call q at different call sites, the pair (p,q) does not define an unique call site. Thus, we add the subscript i to uniquely identify the i^{th} call site from p to q.

$$size\_ratio_q = \frac{size_q}{Total\_application\_size} \tag{3.3}$$

where $freq_{E_i(p,q)}$ is the frequency of the edge $E_i(p,q)$; $freq_q$ is the overall execution frequency of function q in the training execution; $Total\_application\_size$ is the estimated size of the application which is the sum of the estimated sizes of all its functions; $size_q$ is the estimated size of the function q.

$Total\_cycle\_count$ is the estimated total execution time of the application:

$$Total\_cycle\_count = \sum_{k\ \in\ PUset} cycle\_count_k \tag{3.4}$$

where $PUset$ is the set of all program units (*i.e.* functions) in the program, $cycle\_count_k$ is the estimated number of cycles spent on function k.

$$cycle\_count_k = \sum_{i\ \in\ stmts_k} freq_i \tag{3.5}$$

where $stmts_k$ is the set of all statements of function k, $freq_i$ is the frequency of execution of statement i in the training run.

The overall frequency of execution of the callee q is computed by:

$$freq_q = \sum_{k\ \in\ callers_q} freq_{E_i(k,q)} \tag{3.6}$$

where $callers_q$ is the set of all functions that contain a call to q.

Essentially, $cycle\_ratio$ is the contribution of a call graph edge to the execution time of the whole application. A function's cycle count is the execution time spent in that function, including all its invocations. $(\frac{freq_{E_i(p,q)}}{freq_q} * cycle\_count_q)$ is the number of cycles contributed by the callee q invoked by the edge $E_i(p,q)$. Thus, $cycle\_ratio_{E_i(p,q)}$ is the contribution of the cycles resulting from the call site $E_i(p,q)$ to the application's total cycle count. The larger the $cycle\_ratio_{E_i(p,q)}$ is, the more important the call graph edge.

The estimated size of the function q, $size_q$, is computed by:

$$size_q = 5 * BB\_count_q + STMT\_count_q + CALL\_count_q \tag{3.7}$$

where BB_count_q is the number of basic blocks[3] in function q and reflects the complexity of the control flow in the PU, $STMT\_count_q$ is the number of statements in q, excluding labels, parameters, and pragmas. $CALL\_count_q$ is the number of call sites in q.

The $size\_ratio_q$ is the callee q's contribution to the whole application's size.

[3] A basic block is a straight-line piece of code which contains neither branch instructions nor branch targets in the middle.

Finally, the $Total\_application\_size$ is given by:

$$Total\_application\_size = \sum_{k \in PUset} size_k \tag{3.8}$$

The intuition for the temperature heuristic is that edges with high temperature are call-sites that are invoked frequently and whose callee is small compared to the entire application. With careful selection of a threshold on temperature, ORC can find cycle-heavy calling edges whose callee is small compared to the whole application.

Figure 3.1 shows the distribution of the temperature for the bzip2 benchmark.[4] The horizontal axis shows the calling frequency and the vertical axis the temperature. Each dot in the graph represents an edge in the call graph. The temperature varies in a wide range: from 0 to 3000. The calling frequency is shown in reverse order, the most frequently called edges appear on the left side of the graph and the least-frequently-called edges are toward the right side. From left to right, the temperature decreases as the frequency of the call sites also decreases. It is reasonable that the temperature does not go straight down because besides the call-site frequency, the temperature heuristic also takes the callee's size into consideration. Procedure size negatively influences the temperature. Thus, frequently invoked call sites might be "cold" simply because they are too large.

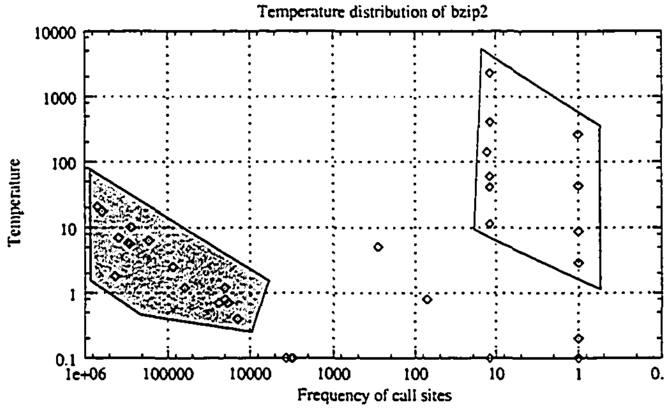


Figure 3.1: Temperature distribution of bzip2

In the original ORC inlining heuristic, a call site is rejected for inlining if its temperature is less than a specified threshold. However, this temperature heuristic may lead to the inlining of edges with high temperature but very low frequency. For instance, we highlighted two clusters of edges in the

[4] To make it easy to read, the two axes of the graphs are drawn using a log scale. Thus some call sites whose frequencies or temperatures are 0 are not shown in the graph. The same situation exists in Figure 3.3.

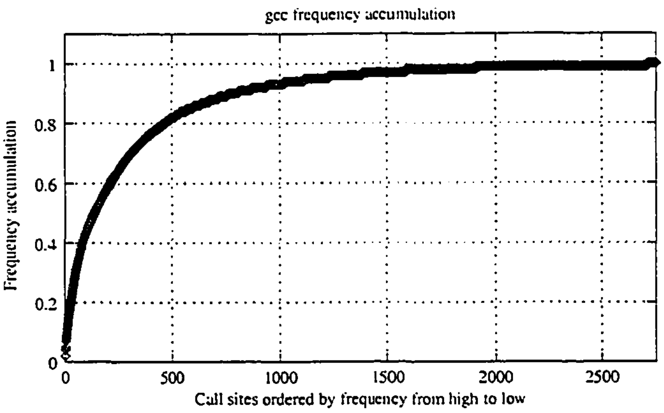


Figure 3.2: Frequency accumulation of gcc (the top 2,750 of all 19,000 call sites are plotted)

temperature×frequency graph for bzip2 in Figure 3.1. The cluster on the right side of the graph has higher temperature but much lower frequency than the cluster on the left side of the graph. Inlining infrequently invoked call sites should always be avoided because it does not help performance. To improve this heuristic, we created a new mechanism to cooperate with the temperature heuristic to prevent the inlining of hot but infrequently invoked call sites. We describe our solution in Section 3.3.

3.3 Inlining Tuning

We improve the inlining heuristics of ORC in two ways. First, adaptive inlining is employed to make the inlining heuristics more flexible. Second, a new *cycle_density* heuristic is introduced to restrict the inlining of hot but infrequent procedures.

3.3.1 Adaptive Inlining

The original inlining heuristic in ORC used a fixed-temperature threshold (120) for inlining decisions. This threshold was chosen as a trade-off among compilation time, executable size, and performance results for different benchmarks. However, a fixed threshold turns out to be inflexible for applications with very different characteristics. For example, a high threshold (*e.g.* 120) is reasonable for large benchmarks because they are more vulnerable to the negative effects of code size increase resulting from inlining. However, the same threshold might not be good for small applications such as mcf, bzip2, and gzip. We will use gcc, which is a typical large application, and bzip2, which is a representative small application, to illustrate this problem.

Figure 3.2 shows the frequency accumulation for the gcc benchmark and Figure 3.3 shows its temperature distribution. In Figure 3.2, the X-axis represents the call sites sorted by invocation frequency from high to low. The i^{th} point numbered from left to right in the figure represents the accumulated percentage of the i most frequent call sites.

gcc has a complex function call hierarchy and the function invocations are distributed amongst a large number of call sites: there are more than 19,000 call sites in gcc. In the standard SPEC2000 training execution of gcc, there are more than 42,000,000 function invocations, and the most frequent call site is called no more than 800,000 times. Figure 3.2 shows that the top 10% (about 2,000) most frequently invoked call sites account for more than 95% of all the function calls. Inlining these 2,000 call sites would result in unbearable compilation cost and substantial code bloat.

In Figure 3.3, according to the frequency of execution, we should inline the call sites on the left hand side of the graph and we should avoid inlining the call sites on the right-hand side. Notice that several call sites on the right-hand side are hot, and thus are inlined by the original heuristics of ORC.

For large applications, the improvement from inlining is usually very limited (as we will see in the Section 3.4). On one hand, it is impossible to eliminate most of the function overheads without wholesale inlining. On the other hand, if we use the same temperature threshold as for small benchmarks, we might end up with the problem of *over-inlining*, *i.e.* too many procedures are inlined and the negative effects of inlining are more pronounced than the positive ones. For example, if the temperature threshold is set to 1, there will be more than 1,700 call sites inlined in gcc. Such aggressive inlining makes the compilation time much longer without performance improvement as our experiments show.

The high temperature threshold (120) in the original ORC was chosen to avoid over-inlining in large applications. However, this conservative strategy impedes aggressive inlining for small benchmarks where code bloat is not as prominent. For instance, Figure 3.1 and Figure 3.4 show the temperature distribution and frequency accumulation of the bzip2 benchmark. There are only 239 call sites and about 3,900 lines of C code in bzip2. This implies that the program is quite small when compared with more than 19,000 call sites and 190,000 lines of C code in the gcc benchmark. Moreover, in bzip2 the top ten most frequently invoked call sites, which comprise about 4.2% of the total number of call sites, accounts for nearly 97% of all the function calls shown in Figure 3.4.

As we will see in the Section 3.4, aggressive inlining is good for small benchmarks such as bzip2. Inlining the 10 most frequently invoked call sites

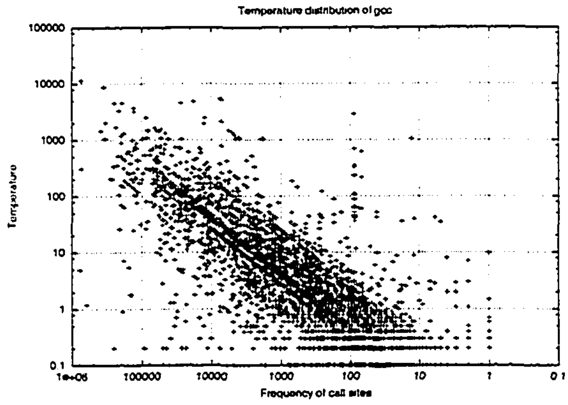


Figure 3.3: Temperature distribution of gcc

in bzip2 eliminates almost all its runtime function calls.

However, the inflexible temperature threshold often prevents the inlining of the most frequent call sites because their temperatures are lower than the fixed threshold (120). Thus, it is desirable that the temperature threshold for small benchmarks be lowered because many of the call sites that have performance potential do not reach the conservative temperature threshold used to prevent code bloat in large applications.

The contradiction between the threshold distributions of large benchmarks and small ones naturally motivates adaptive inlining: we use a high temperature threshold for large applications because they tend to have many hot call sites; and we enable more aggressive inlining for small applications by lowering the temperature threshold for them.

Adapting the inlining temperature threshold according to application size is pretty simple in ORC. Because the estimated size of each procedure in ORC is available in the IPO phase, their sum is the estimated size of the application.[5] We classify applications into three categories: large applications, medium applications and small applications. In the compilation, we utilize proper temperature threshold according to the estimated application size. If an application is a large application, its temperature threshold is 120. If it is a medium application, its temperature threshold is 50. Otherwise, the temperature threshold is lowered to 1. The threshold values were obtained by a detailed empirical study of the SPEC2000 benchmarks. This division of applications into three categories produces better results than any single threshold applied to all benchmarks.

[5] We ignore library functions and dynamic shared-objects because we cannot acquire this information at compilation time.

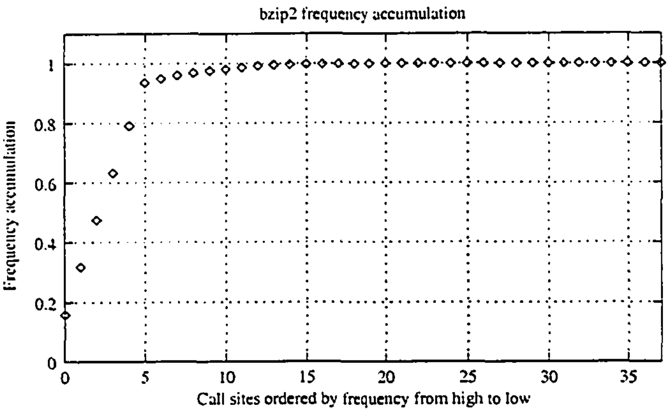


Figure 3.4: Frequency accumulation of bzip2 (the top 38 of all 239 call sites are plotted)

3.3.2 Cycle Density

The intuition behind the definition of temperature is that hot procedures should be frequently invoked and not too large. However, as we have seen in Figure 3.3 and Figure 3.1, some of the procedures with high temperature are not actually hot, *i.e.* some infrequently invoked call sites also have high temperatures. These call sites are represented by the points in the top-right part of the graphs and correspond to functions that are not called frequently, but contain high-trip count loops that contribute to their high *cycle_ratio*, which results in a high temperature (see Equation 3.2). We call the functions that are called infrequently but have high temperatures *heavy functions*.

Inlining heavy functions results in little performance improvement. First, very few runtime function calls are eliminated. Second, the path from the caller to a heavy function is not a hot path at all, and thus will not benefit from post-inlining optimization. Third, inlining heavy functions might prevent frequent edges from being inlined if the code growth budget is spent. To handle this problem, we introduce *cycle_density* to filter out heavy functions.

$$cycle\_density_q = \frac{cycle\_count_q}{frequency_q} \tag{3.9}$$

where $cycle\_count_q$ is the number of cycles spent on procedure q and $frequency_q$ is the number of times that the procedure q is invoked.

When a call site fulfills the temperature threshold, the *cycle_density* of the callee is computed. If the callee has a large cycle count but small frequency, *i.e.* its *cycle_density* is high, it must contain loops with a high trip count. These heavy procedures are not inlined. *cycle_density* has little impact on the performance because it only filters out infrequent call sites. However, using *cycle_density* as a filter can significantly reduce the compilation time and executable sizes, which is important in some application contexts, such as embedded computing.

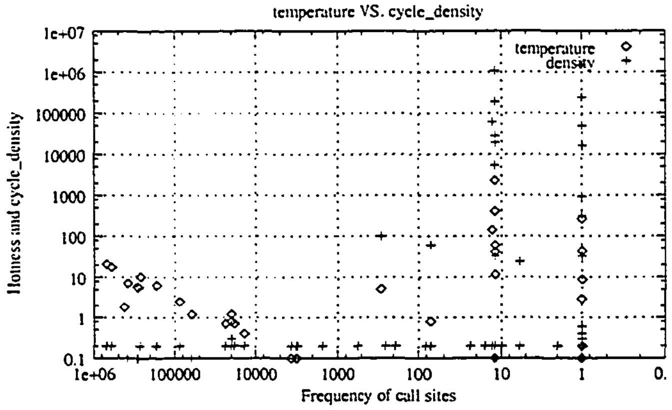


Figure 3.5: Cycle density vs. temperature (bzip2)

Figure 3.5 compares the temperature against the *cycle_density* for each call site in bzip2. For call sites that are actually hot, the temperature is indeed high while the *cycle_density* is low (for bzip2 they are always less than 0.5). These call sites are the ones that will benefit from inlining.

Infrequently-invoked call sites fall into two categories according to their temperatures. Infrequently-invoked call sites with low temperature are eliminated by the temperature threshold. Infrequently-invoked call sites with high temperature always have very high *cycle_density*. Thus we can prevent the inlining of these sites by choosing a proper *cycle_density* threshold. In our tuning, we use a fixed *cycle_density* threshold of 10 that works well for the SPEC2000 benchmarks as we will see in the next section.

We implemented this enhanced inlining decision criteria and contributed it to the ORC-2.0 release. Figure 3.6 shows the C-style pseudo code for the improved inlining analysis in the ORC. Notice that a procedure that has a single call site in the entire application will always be inlined. The reasoning is that the inlining of that single call site will render the callee dead, and will allow the elimination of the callee. Therefore this inlining will save function invocations without causing code growth.

```
InliningAnalysis(CallSite)
    // if this is the only call to the callee. ORC inlines it anyway
    if (CalledOnlyOnce(callee))
        return TRUE
    // MEDIAN_THRESHOLD and LARGE_THRESHOLD are pre-selected thresholds
    // to classify the application as large, small or medium. Accordingly, a proper
    // threshold is selected.
    temperature_threshold ← 120
    if (estimated_size < LARGE_THRESHOLD)
        temperature_threshold ← 50
    if (estimated_size < MEDIAN_THRESHOLD)
        temperature_threshold ← 1
    // temperature(X) computes the temperature of a call site X. The temperature is
    // compared against TEMPERATURE_THRESHOLD to decide whether a call
    // site is hot.
    if (temperature(CallSite) > TEMPERATURE_THRESHOLD
        and cycle_density(CallSite) < CYCLE_DENSITY_THRESHOLD)
            return TRUE
    return FALSE
```

Figure 3.6: Adaptive inlining in ORC

3.4 Results

3.4.1 Experimental Environment

We investigate the effects of adaptive inlining and of the introduction of the *cycle_density* heuristic on performance, compilation time, and the final executable size of SPEC INT2000 benchmarks. We use a cross-compilation method: we run ORC on an IA32 machine (an SMP machine with 2 Pentium-III 600MHz processors and 512MB memory) to generate an IA64 executable which is run on an Itanium machine (733MHz Itanium-I processor, 1GB memory). Thus our performance comparison is conducted on the IA64 systems and our compilation time comparison is conducted on the IA32 system.

3.4.2 Performance Analysis

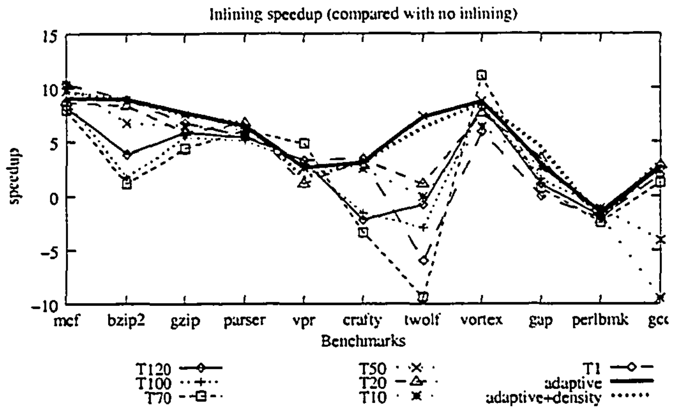


Figure 3.7: Overall performance comparison

Figure 3.7 shows the performance improvement when different inlining strategies are used. T120 represents a fixed temperature threshold of 120, T1 is a fixed temperature threshold of 1, similarly for the other T labels. In adaptive the temperature threshold varies according to the *adaptation* heuristic described in Section 3.2. In the adaptive+density compiler both the adaptation and the *cycle_density* heuristics are used.

Except for perlbmk, in all benchmarks the adaptation heuristic results in positive speedup for inlining. These results suggest that our adaptive temperature threshold is properly selected. In some cases the difference between a fixed threshold and the threshold chosen with adaptation is very significant (see bzip2 and twolf). Note also that the addition of *cycle_density* to adaptation does not produce much effect on performance. This result is explained by the fact that *cycle_density* only prevents heavy and infrequently invoked functions from inlining.

We arranged the benchmarks in Figure 3.7 according to their sizes with the smaller benchmarks on the left and the larger ones on the right. Comparatively, in general, inlining yields better speedups for small benchmarks than for large benchmarks. This observation can be made by examining the maximum performance improvement from all the strategies. Excluding twolf and vortex, the maximum performance improvement decreases from left to right (from small benchmark to large benchmarks). This trend suggests a loose correlation between the application size and potential performance improvements that can be obtained from inlining.

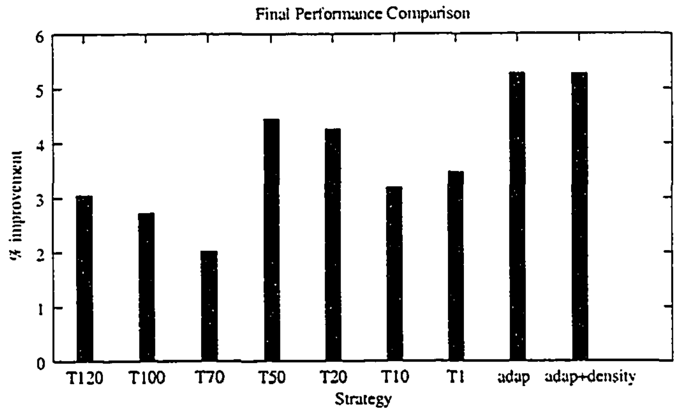


Figure 3.8: Final performance comparison

Figure 3.8 compares the performance improvements of different strategies more explicitly. We first calculate the performance speedup for each benchmark. The baseline is the performance of the 11 benchmarks compiled without inlining. Each bar in Figure 3.8 represents the arithmetic average performance speedup for the 11 benchmarks studied. Finally, the two rightmost bars are for adaptive inlining without and with *cycle_density* heuristics. The adaptive inlining strategy speeds up the benchmarks by 5.28%, while the best average performance gain of all other strategies is 4.45% when the temperature threshold is 50. Notice also that the performance influence of *cycle_density* heuristics is negligible.

3.4.3 Compilation Time and Executable Size Analysis

In this section, we study the effect of the *cycle_density* heuristics on the compilation time and on the executable size. Because *cycle_density* filters procedures that have high temperatures but are infrequently invoked call sites, we expected that their use should reduce both the compilation time and the final executable size.

Table 3.1 shows some statistics collected from different optimization configurations. In the table, "N" means no inlining is used, "A" means adaptive inlining is used and "A-D" means using adaptive inlining plus *cycle_density*. The table first shows the executable size, measured in bytes, and the compilation time, measured in seconds, for all benchmarks when no inlining is performed. Then for the compiler with adaptive inlining and the compiler with adaptive inlining plus *cycle_density*, the table displays the percentage increase in the executable size and compilation time. The table also shows, under the "calls" columns, the number of call sites that were inlined in each

	Executable Size					Compilation Time		
Program	N	A		A-D		N	A	A-D
	(Bytes)	% inc	call	% inc	calls	(Secs)	%inc	%inc
bzip2	116295	54.1	89	26.9	88	70.356	117.8	71.3
gcc	4397983	4.4	919	4.4	919	4194.54	6.0	4.0
crafty	635855	20.1	204	20.1	204	440.687	30.9	30.9
gap	1977644	9.7	345	7.3	343	1409.18	9.1	2.7
gzip	147417	67.6	62	28.0	54	109.457	93.8	41.2
mcf	48241	-0.5	19	-6.3	17	41.832	9.3	8.5
parser	340223	18.1	239	16.4	224	274.868	17.1	12.9
perlbmk	2163047	7.5	419	7.5	419	1518.37	10.6	8.9
twolf	823832	10.6	147	10.6	147	646.769	19.8	20.5
vortex	1170014	31.4	210	31.1	208	1162.27	33.0	36.5
vpr	532912	17.5	141	16.4	139	293.683	30.2	26.2
average		21.9		14.8			34.3	24.0

Table 3.1: Impact of *cycle_density* on executable size and compilation time

case.

The *cycle_density* heuristic significantly reduces the code bloat and compilation time problem. On average, adaptive inlining increases the code size by 21.9% and the compilation time by 34.3%. When *cycle_density* is used to screen out heavy procedures, these numbers reduce to 14.8% and 24%, respectively. It is also interesting to compare the actual number of inlined call sites: the *cycle_density* heuristic only eliminates a few call sites. Except for gzip and parser, *cycle_density* prevents the inlining of no more than 2 call sites in each benchmark. Table 3.1 also shows some curious results. Although *cycle_density* prevents the inlining of a single call site for bzip2, the code growth reduces from 54.1% to 26.9%. A close examination of bzip2 reveals that the procedure *doReversibleTransformation* calls *sortIt* infrequently (only 22 times in the standard training run). However ORC performs a bottom-up inlining, in which the edges in the bottom of the call graph are analyzed and inlined first. In the bzip2 case, *sortIt* absorbs many functions and becomes very large and *heavy* before it is analyzed as the callee. When ORC analyzes the sites that call *sortIt*, the estimated cycle number spent in *sortIt* is huge, which contributes to its high temperature. However, *sortIt* is called infrequently and its inlining does not produce measurable performance benefits. *cycle_density* filters these heavy functions successfully.

Finally, *cycle_density* only eliminates a few call sites because it is not applied to callees that are only called at one call site in the entire application (see Figure 3.6).

3.4.4 Motivation for Partial Inlining

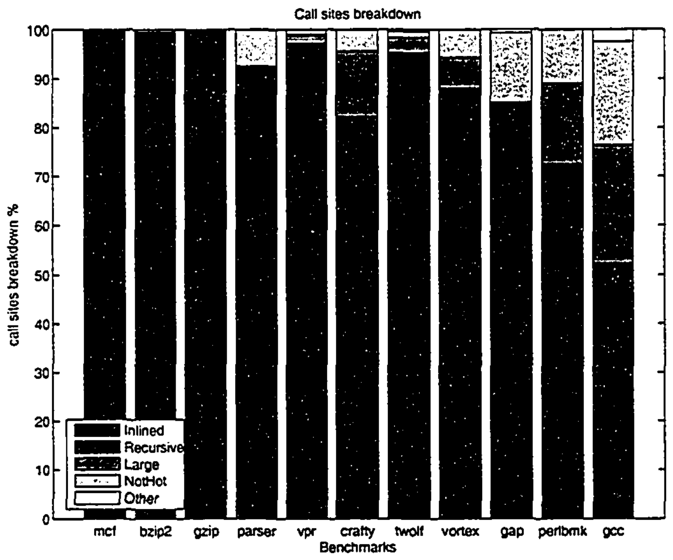


Figure 3.9: Call sites breakdown

Figure 3.9 shows how many dynamic function calls can be eliminated using our adaptive inlining technique. We divided the function calls into five different categories:

Inlined Call sites that can be inlined with our adaptive inlining technique. These call sites have high temperature and low *cycle_density*.

NotHot Call sites that are not frequently invoked. It brings no benefit to inline these call sites.

Recursive ORC does not inline call sites that are in a cycle in the call graph.

Large Call sites that have high temperature but cannot be inlined because either the callee, the caller or its combination is too large. gcc, perlbmk, crafty and gap have some large call sites.

Other Call sites that cannot be inlined due to other reasons. For example, the actual parameters to the call sites do not match the formal parameters of the callee. As Figure 3.9 shows, these call sites are very rare.

With our enhanced inlining framework, we were able to eliminate most of the dynamic function calls for small benchmarks such as mcf, bzip2, and gzip. However, we only eliminated about 30% dynamic function invocations for gcc and 57% for perlbmk. Examining the graph in Figure 3.9, to obtain further benefits from inlining we need to address inlining in these large benchmarks. The categories that are the most promising are the recursive function calls and call sites with large callers or callees.

Figure 3.9 shows that for some large benchmarks (parser, perlbmk, and gcc) a significant portion of the function invocations that are not inlined are recursive functions.

Among the two problems, we are particularly interested in using the idea of code splitting to reduce the sizes of the frequently invoked large functions, which is discussed in Chapter 4.

3.5 Related Work

In this chapter we presented improvements to the inlining heuristics in the Open Research Compiler (ORC). Several researchers have investigated inlining. However, very few of them produced a detailed empirical study using an industry-strength compiler infrastructure based on industry-standard benchmarks such as the one that we present in this thesis.

Ayers *et al.* [7] and Chang *et al.* [22, 47] studied aggressive inlining and cloning. Their inlining facility is very much like that in ORC: the inlining happens on high-level intermediate representation, using feedback information and cross-module analysis. Both of them use a budget to control code bloat: inlining a call site consumes code-growth budget. Ayers *et al.* use an estimated 100% compilation-time increase as their budget for inlining. ORC uses an estimated 100% code-size increase for the inlining budget. In our experiments, inlining in ORC never uses up this budget.

Without feedback information, Allen and Johnson perform inlining at the source code level [4]. Besides reporting impressive speedup (12% on average), they also show that inlining might exert negative impact on performance.

Several researchers have tried to enable aggressive inlining in the context of object-oriented programming. A single call site may have multiple potential callees. For instance, the C and C++ programming languages allow calling functions through pointers. Polymorphism in OO programming languages is often realized via indirect function calls, also called virtual-method invocation. For indirect function calls, it may be impossible to infer the callee before runtime. Thus, inlining cannot be applied straightforwardly to dynamic function calls. A series of special inlining approaches were developed to improve the performance of applications that employ indirect function calls intensively [9, 18, 28, 30, 41].

Chapter 4

Function Outlining

4.1 Introduction

Algorithms used in optimizing compilers are often applied to the scope of a function. Many of these algorithms have super-linear time and spatial complexity on their inputs. Thus compiling a program with large functions demands large memory storage and is time-consuming. Large functions also impose limitations on other optimizations such as function inlining as discussed in Chapter 3. The inlining heuristics used in most compilers avoid inlining call sites that target large callees because inlining large callees causes the *code-bloat problem* [25, 26, 27, 99]. For example, large functions are the most prominent cause that prevent the Open Research Compiler (ORC) from eliminating frequently called sites. In Chapter 3, we have shown that, even after tuning, ORC only eliminates about 30% of the runtime function invocations for gcc and 57% for perlbmk. Moreover, more than 50% of the hot call sites are not inlined because the callee is too large. Large functions also undermine inter-procedural code layout algorithms. For instance, Pettis and Hansen's "closest is best" code layout algorithm tries to place a caller function next to its most-frequently-invoked callee [72]. The intuition is that proximity between a call site and its callee enhances performance. However, if the caller itself is very large, its most frequent callee may still be placed far away from the call site, defeating the code placement heuristic.

Fortunately, not every statement in a frequently called large function is equally important or executed as often as its host function. There are many examples of large but infrequently executed code in hot functions [64, 65]. For instance, only 8.1% of the code in the BSD version of the TCP network protocol implementation is frequently executed [65]. Another good example is the function *regmatch* in perlbmk in the SPEC2000 benchmark suite. *Regmatch* contains a *switch* statement with about 800 lines of C code to handle 57 string matching scenarios. Although these 800 lines of code are evenly distributed through the 57 cases, only 12 cases occur frequently. Splitting cold code out of hot functions, *i.e. outlining*, is a natural solution to overcome the negative impact of mixing codes with heterogeneous execution frequency. The advantages of outlining cold regions of a hot function are at least three-fold:

Enabling Inlining. When a large cold region is outlined from a hot function, the hot function might become small enough to enable its inlining.

Improving Cache Efficiency. Without outlining, the aggregation effect of large cache lines reduces spatial locality. Cold statements may be loaded into the cache when hot statements are loaded. Segregating hot and cold regions into separate functions enables better code placement to improve the cache utilization.

Improving Instruction Fetch Bandwidth. Modern superscalar and VLIW architectures demand high instruction bandwidth of the memory hierarchy. A sufficient number of useful instructions must be fetched into the cache for full utilization of the functional units in the processors. For instance, Mosberger *et al.* found that limited instruction bandwidth results in almost 70% of CPU cycles idle in some architectures [64]. Separating the hot code from the cold code also improves the utilization of instruction-fetch bandwidth.

A negative performance impact of outlining is that extra function calls are introduced to transfer control between the outlined region and the other parts of the program unit. An efficient implementation of outlining should minimize this performance penalty. This chapter describes the following contributions:

- An abstract syntax tree (ABS)-based region formation that efficiently exploits high-level control flow structures and their associated feedback information to differentiate regions with heterogeneous frequencies.

- A proof that the *Optimal Outlining Problem* (OOP) is NP-hard.

- An effective heuristic to analyze the benefit of outlining a region. This heuristic decision weighs the benefit of reducing the host function size against the frequency of the extra function calls introduced. We also describe how to patch the control flow and data flow to preserve program semantics in outlining.

- Because outlining is an early code transformation, it may negatively impact existing downstream optimizations. Our experiments show that more complex alias relationships created by these parameters have a major impact on downstream optimizations and result in the introduction of substantial memory spills. We propose a novel technique, *alias agent*, to disambiguate parameters created to pass to outlined functions from their counterparts in the host function.

- A study of two orthogonal function splitting strategies: (1) *collective* vs. *independent* splitting; and (2) splitting with vs. without alias agent. This study shows that selecting the correct strategy is crucial. Independent splitting with alias agent reduces function sizes significantly while minimizing the performance penalty of outlining.

Section 4.2 introduces the intermediate representation where outlining is implemented and the concept of region. Section 4.3 describes the design and implementation of outlining. Section 4.4 compares the different outlining strategies and reports the performance of partial inlining. Finally, we discuss related work in Section 4.5.

4.2 Background

An important motivation for function outlining is to enable more aggressive inlining, which is a major component of inter-procedural optimization (IPO) in the Open Research Compiler (ORC). ORC performs IPO very early to enable aggressive function-level compilation. It is thus natural to also implement outlining early in this compiler. The outlining analysis and transformation described in this thesis is a transformation of the WHIRL abstract syntax tree of the function affected.

4.2.1 WHIRL Tree Introduction

ORC's intermediate representation has five levels, from *very-high WHIRL* to *very-low WHIRL* [80]. At the higher levels the WHIRL representation of a function is close to the original source code. We implemented outlining on very-high WHIRL where high-level hierarchical control flow constructs — such as *if*, *loop* and *switch* — have not been transformed to flat constructs — such as conditional branches and *goto*s. Thus outlining can take advantage of these hierarchical constructs and their associated frequency information to identify the cold code segments in a single pass through the WHIRL tree.

A contrived function, *HotPU* shown in Figure 4.1, illustrates the WHIRL tree representation. Statements are annotated with their execution frequency obtained from runtime profiling. Assume that *HotPU* is frequently invoked. The shaded code segments or nodes are the cold parts of *HotPU*.

In very-high WHIRL, three control flow constructs may lead to infrequently executed code in a hot function:

```
HotPU //1000
switch (key)
   case 1: ... break; // 500
   case 2: ... break; // 0
   case 3: ... break; // 500
   case 4: ... break; // 0
   default: ... // 0
endswitch
if (i > 100) // 1, if1
   while (1) // 2
      ... // loop body
   endwhile
   return 0; // 1, ER (Early Return)
else // 999
   if ( i == 101) // 0, if2
      return 1; // ER
   else // 999
      i - -;
      return 2; //999, frequent ER
   endif

   printf("1: not touched"); // 0
endif

printf("2: not touched"); // 0
printf("3: not touched"); // 0
```

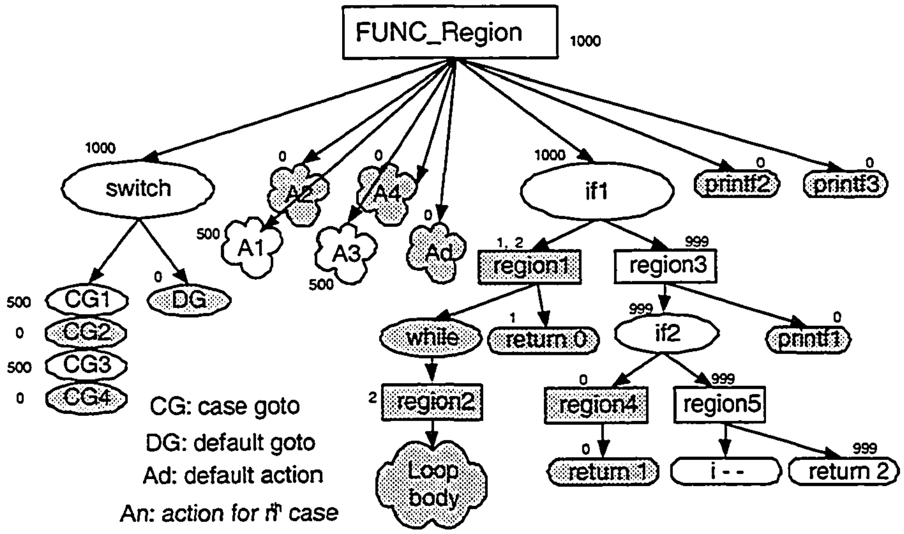


Figure 4.1: Example source code & WHIRL tree

***if* statement.** An *if* node in a WHIRL tree has two children: a *then* block and an *else* block. The feedback information contains the execution frequency of each branch. For example, in Figure 4.1 both *if* statements have skewed execution frequency.

***switch* statement.** In Figure 4.1, each *CG* node corresponds to an enumerated case in a *switch* statement. If the *switch* expression (or key) equals to n, the CG_n node is executed and the program jumps to the A_n node that contains the action code for case n. If the *switch* expression is not equal to any of the enumerated cases, the program jumps to the A_d node that contains the default action through the *DG* node. Feedback information associated with a *switch* statement indicates the execution frequency of each case. Studies have shown that many *switch* statements have skewed execution frequency distribution [100]. In Figure 4.1, only two of the cases in the *switch* statement are hot.

Early return. Early return occurs when the *return* statement or an *exit* function call appears early in a function. Each *return* statement is annotated with its execution frequency. Usually a hot early return implies that the rest of the function is cold. In Figure 4.1, there are three early returns at lines 12, 15 and 18 that correspond to nodes *return*0, *return*1 and *return*2 in the WHIRL tree. However, only *return*2 at line 18 is hot.

Analyzing high-level control flow constructs and their corresponding frequency annotation makes spotting the cold code in a hot function straightforward.

4.2.2 Region

In this thesis, a region is a sequence of code in the program that is guarded by a high-level control-flow construct such as *if* and loop statements (see Figure 4.1). For instance, for an *if-then-else* statement, the code executed under the *then* branch consists of a region and the code executed under the *else* branch forms another region. Likewise, the loop body of a *while* statement is a region.

To handle early returns in the WHIRL representation, we must define the notions of *nearest common ancestor* of two nodes, and of a node's *position* in an ancestor node. A region is represented in WHIRL by a subtree with a BLOCK node as the root. This node has an arbitrary number of children representing the content of the region. The order of a child c in a region R, *order*(c, R), is the one from the source code. In Figure 4.1, *order*(*while*, *region1*) = *1* and *order*(*return0*, *region1*) = *2*. A WHIRL node w has a parent, *parent*(w), and a set of ancestors, *ancestors*(w). In Figure 4.1, *parent*(*while*) = *region1* and *ancestors*(*while*) = {*region1*, *if1*, *FUNC_Region*}. If $s \in R$, R must be an ancestor of s. The position of s in R, $Pos(s, R)$, is given by:

$$Pos(s,\ R) = \begin{cases} order(s,\ R) & if\ parent(s) = R \\ order(s_A,\ R) & if\ (s_A \in ancestors(s)) \wedge (parent(s_A) = R) \end{cases}$$

Thus, if an ancestor of s is a child of R, the position of s in R is the order of that ancestor. In Figure 4.1, we have $Pos(return2, region5) = order(return2, region5) = 2$ and $Pos(return2, region3) = order(if2, region3) = 1$.

Given a node z in a WHIRL tree W, the level of z in W, $Level(z, W)$, is the number of edges that have to be traversed from the root of W to reach z. The root of W is at level 0.

Given a WHIRL tree W and two nodes $y \in W$ and $z \in W$, the ***nearest common ancestor*** of y and z, *NCA(y, z)* is a WHIRL tree node $s \in W$ such that all the following conditions are true:

1. $s \in ancestors(y) \wedge s \in ancestors(z)$
2. $\nexists\, t \in W, (t \neq s) \wedge (t \in ancestors(y)) \wedge (t \in ancestors(z)) \wedge$
   $(Level(t,\ W) > Level(s,\ W))$

An early return statement short-circuits the rest of the current function. However, the short-circuited code might reside in different levels and different regions in the WHIRL tree. For example, *return2* leads to three unexecuted *print* statement: *printf1* in *region3*; *printf2* and *printf3* in *FUNC_Region*. The code short-circuited by an early return *er* in region R, $SC(er, R)$ is defined by:

$$SC(er,\ R) = \left\{ s \,\middle|\, \big(A = NCA(er, s)\big) \wedge \big(Pos(s, A) > Pos(er, A)\big) \right\}$$

In the example, *SC(return2, region3)* includes *printf1* and *SC(return2, FUNC_Region)* includes *printf2* and *printf3*.

4.3 Function Outlining

There are three phases in function outlining optimization: *region reorganization* transforms the WHIRL tree so that the cold code is in separated regions from hot code; *candidate identification* identifies regions for which outlining is beneficial; *function splitting* generates a new function from a candidate region and replaces the region with a call to the new function.

In biased *if* statements, the hot code and the cold code are well structured in two separate sub-regions. However, for *switch* statements and early returns, hot and cold codes are mixed with each other. A depth-first pass on the WHIRL tree reorganizes codes with heterogeneous execution frequency due to *switch* statements and early returns into different regions for convenient splitting.

4.3.1 Reorganize a *Switch* Statement

In most modern programming languages multi-way branch semantics are expressed by *switch* statements. These statements are frequently used in the implementation of script interpreters, compilers, and virtual machines because these applications often use the value of a key to select one among a large collection of possible actions.

A *switch* statement contains a *key expression*, a set of (*case value, case action code*) pairs and a *default action code.* The execution of a *switch* statement has two distinct phases: case selection and case action [78]. Case selection decides which case should be executed based on the value of the key (or *switch* expression). If the key is equal to one of the enumerated case values, the control flow is directed to the corresponding action code. Otherwise, the default action code is executed. In this case, we say that the key falls in the default category or that it is a default case.

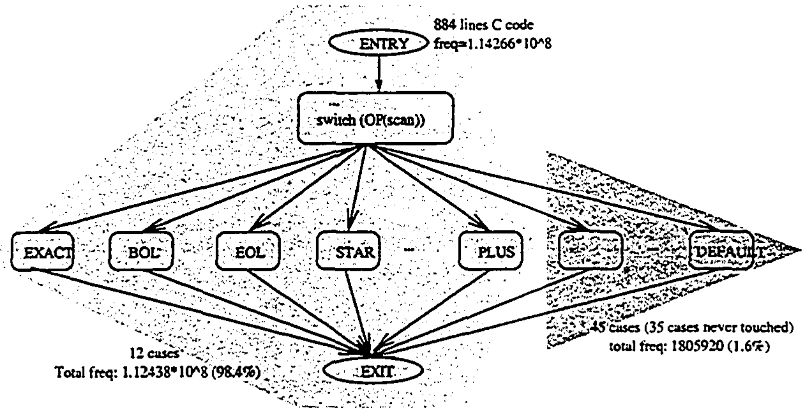


Figure 4.2: Annotated control flow graph of function *regmatch* in perlbmk

To write robust application programs, one must handle many boundary cases, even the ones that seldom occur during program execution. As a consequence, the execution frequency of action cases is often skewed with a small fraction of cases dominating the execution of a *switch.* For instance, Figure 4.2 shows the breakdown of the cases in a *switch*, S, in the function *regmatch* of the SPEC2000's perlbmk program. This function matches regular expressions with strings in a file. S requires about 884 lines of C code evenly distributed through 57 cases. But only 12 cases are hot. This is because some expres-

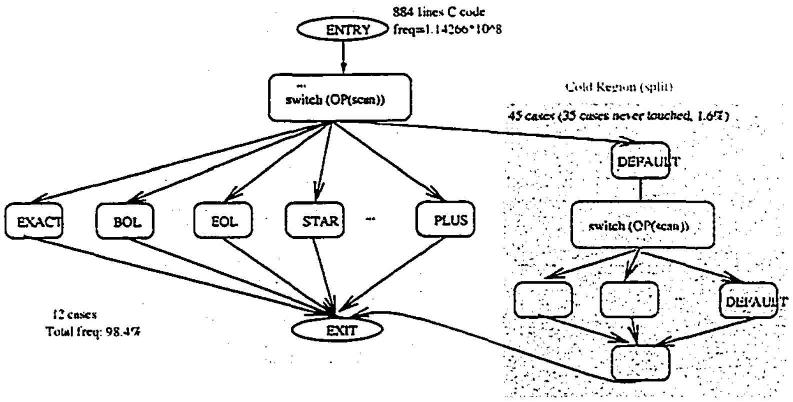


Figure 4.3: Partitioning *regmatch* in `perlbmk`

sion matchings, such as exact matching (EXACT) and matching zero, one, or more times (STAR and PLUS), occur often. However, exotic matchings, such as matching a string from n to m times and matching a string backward, seldom occur at runtime.

Cold cases introduce several problems: (1) they increase the size of the function that hosts the *switch*, which prevents inlining; (2) the code for cold-case actions, which is intertwined with the hot case actions, pollutes the instruction cache; (3) cold cases may slow case selection by increasing the depth of the comparison tree or cause inefficient usage of memory by the slots for cold cases in the jump-table. Separating the cold cases and their actions from the hot cases ameliorates all these problems.

Switch partitioning first partitions a large *switch* S into two: a hot *switch* S_h containing the hot case selections and the hot action code, and a cold *switch* S_c with the remainder cases and code of S. After this reorganization, a new, simple, and fast, tree-based splitting technique can split S_c out of the host function. The combination of *switch* reorganization and splitting elegantly solves the problems caused by cold cases: (a) the host function becomes smaller and is more amenable to inlining; (b) the hot cases are placed together without pollution from cold cases at runtime; and (c) the execution path for the selection of hot cases becomes shorter.

Figure 4.3 shows the partition of S into S_h and S_c in *regmatch*. The cold cases are clustered into S_c and placed into the default action of S_h. After the reorganization, S_c forms a natural cold region and can be easily split out of the host function. Not all large *switch*es can be partitioned in the way shown in Figure 4.3, see Section 4.3.1.3 for details.

4.3.1.1 Prepass for *Switch* Reorganization

In preparation for *switch* reorganization, a prepass summarizes the feedback information, and clusters cases based on control flow information.

(i) Summarize the *switch*.

The total frequency of a *switch* S is:

$$S.total\_freq = \sum_{i=1}^{S.num+1} S.Freq[i] \tag{4.1}$$

where $S.num$ is the number of enumerated cases in S and $S.Freq[S.num+1]$ is the frequency of the default case of S.

(ii) Cluster cases according to control flow information.

If there is no transfer of control between different cases in S, identifying hot cases and cold cases is the result of a simple analysis of the feedback information. However, inter-case control flow is often found in application programs. For instance, a common programming trick is to let control fall through from case i to case $i+1$. Moreover, an action may end with an explicit *goto* that transfers control to an arbitrary label in the program.[1] Therefore, the feedback information alone is not sufficient to identify hot actions. Consider a program where the hot action of a case A falls through to the action of another case B. The action of case B is also hot, even though its frequency in the feedback information may be low. In this circumstance, A and B must be an atomic unit for the splitting analysis.

A *goto* transfers control from a *source* location to a *destination* location. Given a *switch* S, we say that a *goto* g is in S, $g \in S$, if both the source and destination locations are within S. If $g \in S$, we refer to a $g.source\_case$ and to a $g.destination\_case$. The algorithm PREPASS, shown in Figure 4.4, computes case groups based on control flow. First, each case is assigned to a distinct group ranging from 1 to $S.num+1$ (steps 2-3). Two situations are of interest: fall-through between adjacent cases (steps 4-7) and *goto*s that are in S and whose source and destination cases are distinct (steps 8-10).

MERGEGROUPS, called by PREPASS, merges two case groups into one. Whenever two groups are merged, their members are assigned the same new *merged_group_id* (step 5). MERGEGROUPS also calculates the execution frequency of the merged group, which is the sum of the invocation frequency of its member cases (step 6).

4.3.1.2 *Switch* Partitioning Benefit Estimation

After the prepass phase, the compiler analyzes the summarized information to decide whether a *switch* should be split. If the decision is yes, the compiler

[1] This situation appears frequently in heavily hand-optimized applications such as `perlbmk`.

```
PREPASS($S$)
$merged\_group\_id \leftarrow S.num + 2$;
foreach $i$ from 1 to $S.num + 1$;
        $case[i].group \leftarrow i$;
foreach $i$ from 1 to $S.num$
        if ($case[i]$ falls through to $case[i + 1]$)
          then MERGEGROUPS($S, i, i + 1, merged\_group\_id$);
                $merged\_group\_id \leftarrow merged\_group\_id + 1$;
foreach $g \in S$ such that $g.source\_case \neq g.destination\_case$
        MERGEGROUPS($S, g.source\_case, g.destination\_case, merged\_group\_id$);
        $merged\_group\_id \leftarrow merged\_group\_id + 1$;


MERGEGROUPS($S, i, j, new\_id$)
$S.Freq[new\_id] \leftarrow 0$;
$group\_i \leftarrow case[i].group$;
$group\_j \leftarrow case[j].group$;
foreach $k$ such that $case[k].group = group\_i$ or $case[k].group = group\_j$
        $case[k].group \leftarrow new\_id$;
        $S.Freq[new\_id] \leftarrow S.Freq[new\_id] + S.Freq[k]$;
```

Figure 4.4: Case clustering

also find the cases that should be split out of the original *switch*. The input for this analysis is the *switch* S annotated with the groups formed by the prepass phase.

PARTITIONANALYSIS, shown in Figure 4.5, sorts the case groups according to their frequencies from high to low. Then the algorithm scans the case groups and accumulates their execution frequency. When the accumulated frequency reaches *Freq_Threshold* (99% in our work), the scanning stops. The decision to split the cold cases is based on the size of the cold cases. If *ColdSize* is larger than a set threshold (40 in our work), the *switch* is reorganized into a hot *switch* and a cold *switch* as illustrated in Figure 4.3, and the cold *switch* can be split out of the host function now.

```
PartitionAnalysis($S$)
$AccuFreq \leftarrow 0$;
$HotSize \leftarrow 0$;
$HotGroups \leftarrow \emptyset$;
foreach non-empty group $i$ from the most frequent to the least frequent
        $AccuFreq \leftarrow AccuFreq + S.Freq[i]$;
        $HotSize \leftarrow HotSize + S.Size[i]$;
        if $\left(\frac{AccuFreq}{total\_freq} \leqslant Freq\_Threshold\right)$
          then $HotGroups \leftarrow HotGroups \bigcup i$;
        else break; // terminate the loop
$ColdSize \leftarrow S.total\_size - HotSize$;
if ($ColdSize > Size\_Threshold$)
  then $ColdSwitch \leftarrow$ PartitionSwitchCaseStmt($S, HotGroups$);
```

Figure 4.5: Partition benefit analysis

4.3.1.3 Partitioning *Switch*es With Hot Default

Not all large *switch*es can be reorganized in the way shown in Figure 4.3. When the *default* case is seldom executed, as is the case in Figure 4.3, the original *default* is simply placed into the cold *switch*. However if the original *default* case is frequently executed, moving it into the cold *switch* is troublesome for two reasons: (1) a hot case is still mixed with the cold cases; and (2) if splitting is applied, many additional function calls will occur at runtime. Therefore two classes of *switch*es must be treated separately: *hot-default* and *cold-default switch*es. For *hot-default switch*es the reorganization shown in Figure 4.6 works well. The main difference with the reorganization illustrated by the example in Figures 4.2 and 4.3 is that now the default action remains in the hot *switch*. Also the case selection of cold cases is kept intact. The actions of cold cases are replaced with *goto*s that transfer control to the new cold *switch*. The original cold actions are *moved* into the cold *switch*. After this transformation, if the *switch* expression contains a cold value, its case selection requires: (1) the original case selection; (2) a function call; and (3) a second case selection in the cold *switch*. If the cold cases are indeed cold, this additional function call will happen infrequently and thus have a negligible cost.

PartitionSwitchCaseStmt, shown in Figure 4.7, partitions a *switch* into two and then splits the new cold *switch* into a new, independent region.

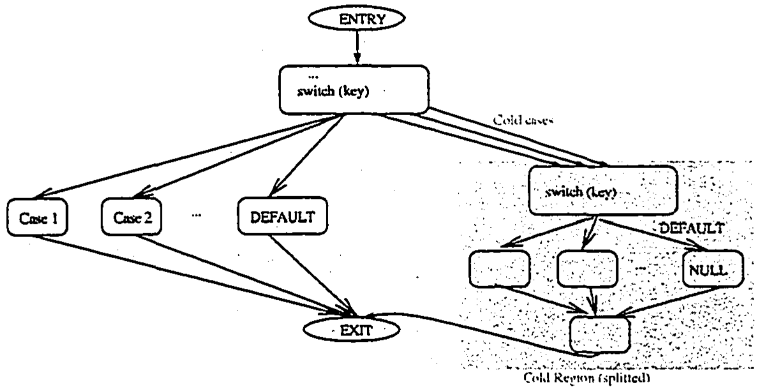


Figure 4.6: Partitioning a *switch* with hot default cases

```
PartitionSwitchCaseStmt($S$)
if ($C_d$ is hot)
    $ColdSwitch \leftarrow$ CreateSwitch($S.ColdCases$, NULL);
    $NewRegion \leftarrow$ BuildRegion($ColdSwitch$);
    foreach ($C_i$, $A_i$) in $S.ColdCases$
            $Goto_i \leftarrow$ NewGoto($NewRegion$);
            Replace($A_i, Goto_i$);
else
    $OrigDefault \leftarrow (C_d, A_d)$;
    $NewColdCases \leftarrow S.ColdCases - OrigDefault$;
    $ColdSwitch \leftarrow$ CreateSwitch($NewColdCases, OrigDefault$);
    $NewRegion \leftarrow$ BuildRegion ($ColdSwitch$);
    foreach ($C_i$, $A_i$) in $S.ColdCases$
            Delete($C_i$, $A_i$);
    Replace($A_d, NewRegion$);
RepairFeedbackInformation($S$);
return $ColdSwitch$;
```

Figure 4.7: Partition and split *switch*

The algorithm actions are different for cold and hot default situations. In the algorithm, C_i represents case i and C_d is the default case; A_i is the action code for case i and A_d is the default action code. CREATESWITCH generates a new *switch*, *ColdSwitch*, containing the cases listed on CREATESWITCH's first parameter and the default case that appears in CREATESWITCH's second parameter. When the default case is hot (steps 1-6), the default action of the cold *switch* is empty. REPLACE replaces the actions of the cold cases in the original *switch* with *goto*s to the newly generated region.

When the default case is cold (steps 8-14), the cold *switch* is created with the cold cases, including the cold default. This cold *switch* is split into *NewRegion*. Then the cold cases and their action codes are deleted from the original *switch*. The default action of the original default case is replaced with *NewRegion*.

4.3.2 Handling Frequent Early Returns (ER)

The algorithm HANDLEER, shown in Figure 4.8, handles early returns. HANDLEER is called when an early return statement S_{er} is encountered during the depth-first traversal of the WHIRL tree. *ReturnFreq* accumulates the execution frequency of early returns (step 1). Its value is reset to zero before the scan of a function starts, and is preserved between calls to HANDLEER. Unless S_{er} resides in a loop body, when the ratio between the accumulated frequency and the frequency of the host function reaches the *ERThreshold*, the code after S_{er} is cold. If S_{er} is inside a loop body S_{loop}, it is possible that the code in $SC(S_{er}, S_{loop})$ is still hot and we avoid outlining it. We use an upward traversal from the early return S_{er} to find its uppermost loop ancestor S_{loop} (step 6-9). If there is no loop ancestor, S_{loop} is set to be S_{er} itself. The cold code resulted from frequent early return is $SC(S_{loop}, FUNC\_BODY)$. The cold code might spread into different levels of the WHIRL tree (*e.g.* the three *printf* statements in our example). To preserve program correctness, code from different levels cannot simply be put together in a single region. Instead, an upward traversal from S_{er} (step 11-15) extracts the cold code of every region that it encounters into a new region (step 14).

4.3.3 Outlining Candidate Identification

After region reorganization, every cold code snippet is placed in an independent region and annotated with $(frequency, size)$. The *size* of a region is the number of WHIRL nodes in that region. Next the compiler identifies cold regions that are suitable for outlining.

```
HANDLEER ($S_{er}$)
$ReturnFreq \leftarrow ReturnFreq + GetFreq(S_{er})$
if $\left(\frac{ReturnFreq}{GetFreq(HostFunc)} \leqslant ERThreshold\right)$
    return
$S_{loop} \leftarrow S_{er}$
$CurrentParent \leftarrow GetParent(S_{er})$
while ($CurrentParent \neq ROOT$) // $ROOT$ is the root of the WHIRL tree.
    if ($CurrentParent$ is a loop construct)
        $S_{loop} \leftarrow CurrentParent$
    $CurrentParent \leftarrow GetParent(CurrentParent)$
$CurrentNode \leftarrow S_{loop}$
while ($CurrentNode \neq ROOT$)
    $CurrentParent \leftarrow GetParent(CurrentNode)$
    if ($CurrentParent$ is a region)
        call EXTRACTCOLDCODEINTOREGION($SC(CurrentNode, CurrentParent)$)
    $CurrentNode \leftarrow CurrentParent$
```

Figure 4.8: Handling early exits

4.3.3.1 Hazardous program units for outlining

Some program units are not outlined to prevent performance degradation or to preserve program correctness. Besides trivial functions that are rarely executed, the following are not outlined.

Small regions. Outlining replaces a region in a host function, f_{host}, with a function call to a new function, f_{out}. Code patches are often required, before and after the call to f_{out}, to preserve correctness. If a region is too small, these patches might be larger than the outlined region. This kind of outlining is strictly avoided because it fails to reduce the size of the original program unit.

Regions with escaped *alloca*-allocated memory. *Alloca* allocates memory space in the stack frame of a function. This memory is automatically freed when the function returns. When a function uses *alloca* to allocate memory in a region and references the allocated memory outside of the region, the region should not be outlined. This is because f_{out} would allocate a memory block with *alloca* and pass this block to f_{host}. It would be difficult to maintain the original semantics of the program because the memory allocated in f_{out} would be automatically freed at its exit and would be no longer valid in f_{host}.

4.3.3.2 Optimal Outlining Problem is NP-hard

Cold regions are not always beneficial for outlining. The major benefit of outlining is the size reduction of the host function that enables more aggressive inlining and improves code layout. Splitting a segment of code out of a function has several costs. First, necessary code patches may eliminate the size reduction benefit. Second, because the original cold region in f_{host} is replaced by a function call to f_{out}, there is a performance penalty to execute the cold region. It is a hazard to outline hot code because it will result in many runtime function calls. Therefore, an *Optimal Outlining Problem* (OOP) can be formulated as a constrained optimization problem: f_{host} is formed by a set of regions. Assume precise frequency F_i and size S_i information for each region R_i in f_{host}, and a given budget of extra runtime calls K, find a set of regions that, when outlined, minimizes the size of f_{host} without exceeding K. The 0-1 knapsack problem [37] can be reduced to OOP. Consider N items $\langle v_i, w_i \rangle$ where each item has value $v_i > 0$ and weight $w_i > 0$. The 0-1 knapsack problem is the problem of finding a vector with N binary elements d_i that satisfy the following condition:

$$Maximize(\sum_{i=1}^{N}(d_i \times v_i)) \text{ such\_that } \sum_{i=1}^{N}(d_i \times w_i) \leqslant K \text{ } and \text{ } d_i \in \{0, 1\} \quad (4.2)$$

Given a knapsack with capacity K and N items in the 0-1 knapsack problem, we construct a corresponding OOP instance as follows. Each item $\langle v_i, w_i \rangle$ represents a region with size v_i and outlining cost w_i. Let K be the extra runtime call budget. The conversion complexity is linear on the number of items N. Therefore, the 0-1 knapsack problem can be reduced to OOP in linear time. The 0-1 knapsack problem is a well-known NP-hard problem, therefore OOP is also NP-hard. Thus, a reasonable heuristic to find approximate solutions to OOP is necessary.

4.3.3.3 Engineering Approach to Selective Outlining

Given a region R_i in a function f_{host} with frequency F_i and size S_i, we define the frequency ratio of R_i and the size ratio of R_i as:

$$size\_ratio(R_i) = \frac{S_i}{size(f_{host})} \quad (4.3)$$

$$freq\_ratio(R_i) = \frac{F_i}{frequency(f_{host})} \quad (4.4)$$

We adopt a popular *knapsack problem* greedy algorithm to estimate the benefits of splitting a region R_i out of f_{host} [62]. The greedy algorithm has tight time and space bounds and has been proved to be effective in many cases. The idea is to calculate the profit density for each region. In this framework the profit density of a region is called *benefit*:

$$benefit(R_i) = \frac{size\_ratio(R_i)}{freq\_ratio(R_i)} \tag{4.5}$$

The regions are then sorted in decreasing order of their *benefit* value. Regions are selected for outlining, equivalent to placing items into a knapsack, until the constraint in equation 4.2 is violated, *i.e.* the knapsack cannot hold any more items.

Essentially, $size\_ratio(R_i)$ and $freq\_ratio(R_i)$ estimate the contribution of R_i to the total size and execution frequency of f_{host}. Therefore, this heuristics favors large cold regions. Intuitively, larger regions that are not executed frequently should produce the most benefit from outlining and incur the least runtime penalty.

To avoid a situation in which the patch code is larger than the outlined region, there is a threshold for the size of a region R to be outlined:

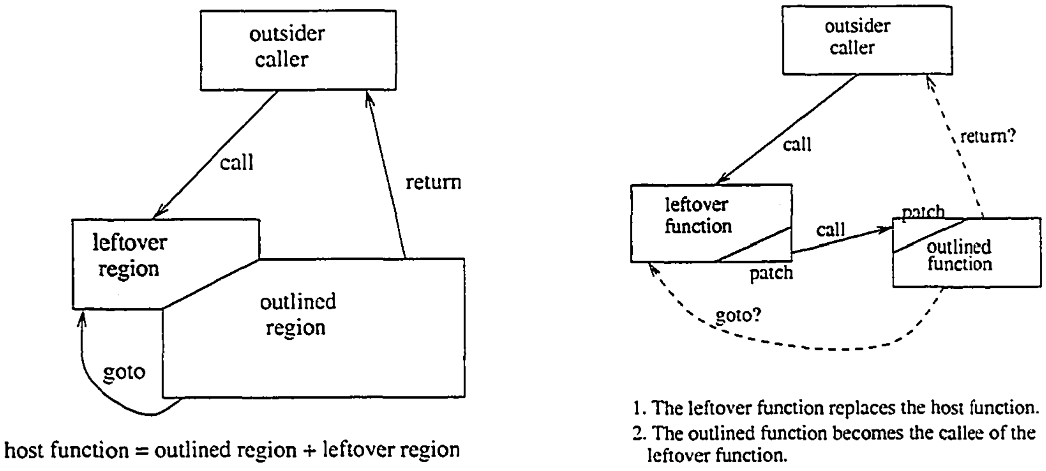
$$size_R > SizeThreshold \tag{4.6}$$

4.3.4 Function Splitting and Patching

In this discussion of outlining we adopt the following terminology: f_{host} (*host function*) is the original program unit in which a region is selected for outlining; R_{out} (*outlined region*) is the region within f_{host} that is selected for outlining and $R_{leftover}$ (*leftover region*) is f_{host} excluding R_{out} (Figure 4.9.a); f_{caller} (*outsider caller*) is a function that calls f_{host}; f_{out} (*outlined function*) is the new function that is generated by the outlining process to contain R_{out} and $f_{leftover}$ (*leftover function*) is the original program unit after R_{out} is split out of f_{host}. After outlining (Figure 4.9.b), f_{out} becomes the callee of $f_{leftover}$ and f_{host} is replaced by $f_{leftover}$ in the call chain. $f_{leftover}$ inherits all the original resources of f_{host}, including the function name, patched WHIRL tree and symbol table.

The outlining transformation consists of three major phases. *Summarizing* collects information that is needed by function splitting. *Callee generation* generates the f_{out} program unit for the outlined region R_{out}. *Caller patching* eliminates the split code and inserts compensation code in f_{host} to conserve the correct semantics. Function *foo* in Figure 4.10 will be used as an example of an f_{host} program unit to demonstrate the outlining process.

Assume that the identification algorithm determines that the shadowed code in Figure 4.10 is a region to be split (*i.e.* R_{out}).


outsider caller
call
return
leftover region
outlined region
goto
host function = outlined region + leftover region
outsider caller
call
return?
leftover function
call
patch
outlined function
patch
goto?
1. The leftover function replaces the host function.
2. The outlined function becomes the callee of the leftover function.

(a) Before outlining

(b) After outlining

Figure 4.9: Outlining transformation

4.3.4.1 Summarizing

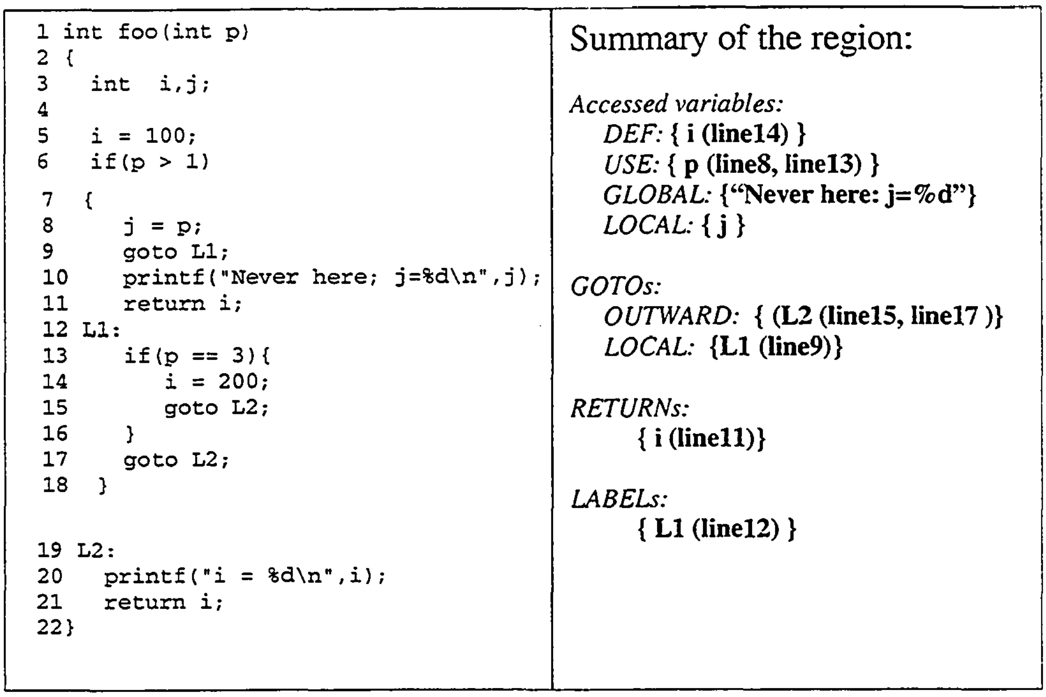
The summary information of a region is used in the outlining transformation. This step traverses the WHIRL representation of the region and collects information including:

i. Variable access information. There are two kinds of variable accesses: *use* and *definition*. A variable's use reads the value of the variable and a variable's definition writes a new value to the variable. For example, in the statement $a = x + y$, variables x and y are used and variable a is defined. The variable access information is used to determine the variables that must be passed to the new program unit.

There is no need to collect access information for global variables because they can be seen by all the functions, including the newly generated program unit.

The right column of Figure 4.10 lists the variables accessed by *foo*. The first parameter to the $printf$ function call is a global symbol representing a constant string and is accessible by all the functions, it does not need to be passed as a parameter to f_{out}. Note that the variable j is never accessed outside of R_{out}. Thus, j can be converted to a local variable in f_{out} to avoid the overhead of passing it as parameter.

ii. Local label information. We collect all the labels that appear in R_{out}. This is because labels are also symbols in a function. After the outlining transformation, the labels in f_{host} are converted to labels in f_{out}. In Figure 4.10, there is one local label $L1$ in R_{out}.



```
1 int foo(int p)
2 {
3   int  i,j;
4
5   i = 100;
6   if(p > 1)
7  {
8     j = p;
9     goto L1;
10    printf("Never here; j=%d\n",j);
11    return i;
12 L1:
13    if(p == 3){
14       i = 200;
15       goto L2;
16    }
17    goto L2;
18  }

19 L2:
20   printf("i = %d\n",i);
21   return i;
22}
```

Summary of the region:

Accessed variables:
DEF: **{ i (line14) }**
USE: **{ p (line8, line13) }**
GLOBAL: **{"Never here: j=%d"}**
LOCAL: **{ j }**

GOTOs:
OUTWARD: **{ (L2 (line15, line17)}**
LOCAL: **{L1 (line9)}**

RETURNs:
{ i (line11)}

LABELs:
{ L1 (line12) }

Figure 4.10: Function *foo* before function splitting

iii. ***Goto* and *return* information.** We divide *goto* statements in R_{out} into two categories: outward *gotos* and intra-regional *gotos*. Outward *gotos* are the *goto* statements that jump to a label in $R_{leftover}$. A *goto* statement in R_{out} that jumps to a label also in R_{out} is an intra-regional *goto.*

It is easy to tell whether a *goto* statement is outward or intra-regional by checking whether the destination of the *goto* falls in the local label list.

Return statement information is used to insert compensation code in f_{host} and f_{out} so that when *return* is executed, f_{out} returns to f_{caller}, *i.e.* the caller of the f_{host}.

Figure 4.10 shows the *gotos*, *returns* and *labels* found in the R_{out} region of *foo.*

4.3.4.2 Outline the region

This step involves generating f_{out} based on R_{out}. Figure 4.11 shows $f_{leftover}$ and Figure 4.12 shows the new function f_{out} :$fooNEW1$. Function splitting needs to perform the following tasks:

i. **Construction of f_{out} and its symbol table.** The compiler needs to build a valid WHIRL tree and its symbol table for f_{out}. The compiler first

```
int foo(int p)
{
  int  i,j;
  int ReturnFlag, ReturnValue;
  i = 100;
  if(p > 1)
  {
      fooNEW1(&ReturnFlag, &ReturnValue, &i, p);

      if (ReturnFlag != 0) {
          if (ReturnFlag == 1)
             return (ReturnValue);
          else
             COMPGOTO(ReturnFlag-2,{L2} );
      }
   }

L2:
   printf("i = %d\n",i);
   return i;
}
```

Figure 4.11: The original *foo* function after function splitting

```
void
fooNEW1 (int *ReturnFlag, int *ReturnValue, int *iNEW, int pNEW)
{
      int  j;
      *ReturnFlag = 0;
      j = pNEW;
      goto L1NEW;
      printf("Never here; j = %d\n", j);
      *ReturnValue = *iNEW;          // return I;
      *ReturnFlag = 1
      return;
L1NEW:
      if( pNEW == 3 ){
          *iNEW = 200;               // i = 200;
          *ReturnFlag = ( 0 + 2 ); // goto L2
          return;
      }
      *ReturnFlag = ( 0 + 2 );     // goto L2
      return;
 }
}
```

Figure 4.12: The new *fooNEW1* function after function splitting

builds an empty WHIRL tree and then clones the WHIRL tree of R_{out} into the empty WHIRL tree. This newly generated WHIRL tree is not valid because both f_{host} and f_{out} need to be repaired to interact properly. Also, the symbol table for f_{out} is initialized.

ii. **Patching variable accesses.** Because R_{out} is only a part of the original program unit, all the variables accessed in R_{out} are within the scope of f_{host}. Thus, when R_{out} is transformed into f_{out}, the scope of the variable accesses must be modified to access the correct memory location.

In very high WHIRL representation, variable accesses fall into four categories:

- Load (LOAD) a variable
- Store (STORE) a variable
- Load address (LDA) of a variable (*i.e.* address taking)
- Indirect load and store (ILOAD and ISTORE)

In the WHIRL representation, ILOAD and ISTORE never directly access a variable in the program. Instead, their operand is a LOAD statement or another ILOAD statement. Therefore, we only need to take care of LOAD, STORE and LDA cases.

If a local variable v is accessed in both R_{out} and $R_{leftover}$, we need to pass v as a parameter to f_{out}. In Figure 4.10, we only need to pass the variables in the DEF and USE groups. A variable can be passed by its value or by its address, depending on whether it is defined in R_{out}. If the variable is defined in R_{out}, we place it in the DEF group and pass its address to f_{out}. Otherwise it is in the USE group and its value is passed.

Because the original WHIRL tree accesses the variables directly, some variable access nodes in the WHIRL tree of R_{out} must be patched according to the parameter passing scheme used for f_{out}. Table 4.1 lists the data access patching rules. In this table, $\boldsymbol{a}$ is the variable in f_{host}, $\boldsymbol{A}$ is the parameter to f_{out}, "–" represents an invalid situation that should not appear in a correct transformation. The binding column specifies how the variable $\boldsymbol{a}$ should be binded to the formal parameter $\boldsymbol{A}$. In the three rightmost colums of the table, the first row is the original variable access statement in f_{out}. The second and the third rows list the transformation needed when a variable is passed by its value and by its address, respectively.

iii. **Building local labels and local *goto* statements** The compiler converts the labels in R_{out} to local labels in f_{out}. Essentially, it generates a new

	passing method	binding	LOAD a	STORE a	LDA a
USE	by value (*i.e.* a)	$A = a$	LOAD A	–	–
DEF	by address (*i.e.* $\&a$)	$A = \&a$	ILOAD A	ISTORE A	LOAD A

Table 4.1: Variable patching rule

label in the local symbol table of f_{out} for each label in R_{out} and modifies every intra-regional *goto* statement to jump to the respective new label. In Figure 4.12, the original label *L1* in *foo* is changed to a local label $L1NEW$ in function $fooNEW1$, and the original intra-regional *goto* statement is modified to point to label $L1NEW$ (Line 7 and 12 in Figure 4.12).

iv. **Handling function exit.** Function f_{out} returns to $f_{leftover}$ in one of three ways:[2]

1. The control naturally falls through the new function and returns to $R_{leftover}$. This kind of returning implies that the next operation executed upon the return is the WHIRL node next to R_{out} in the WHIRL tree of f_{host}.
2. A *return* statement is executed. Originally, the *return* statement returns from f_{host} to its caller f_{caller}. After the outlining, f_{out} becomes the callee of $f_{leftover}$. Therefore, we need to develop a mechanism that returns from f_{out} to f_{caller}.
3. An outward *goto* statement is executed. An outward *goto* means that the control needs to be directed from f_{out} to the specified label in $f_{leftover}$.

We implement the transfers of control by creating two new variables in the original program unit: *ReturnFlag* and *ReturnValue* (Line 4 in Figure 4.11). The addresses of these symbols are both passed to f_{out} as parameters. The integer variable *ReturnFlag* is set by f_{out} as a flag to specify the action that should be taken by $f_{leftover}$ upon the return of f_{out}.

Table 4.2 shows $f_{leftover}$'s action according to *ReturnFlag* on f_{out}'s return. When *ReturnFlag* is 0, the next instruction to R_{out} is executed. When *ReturnFlag* equals 1, the $f_{leftover}$ returns to its caller immediately, *i.e.* f_{out} returns to its caller's caller.

[2] A program unit can also return by the *exit*() function call, but it will exit the application and we do not need to worry about it.

ReturnFlag	Action
0	fall through
1	return *ReturnValue*
≥ 2	computed goto (*ReturnFlag*-2, JUMPTABLE)

Table 4.2: Semantics of *ReturnFlag* on the return of the new PU

In $f_{leftover}$, we build one jump table for each region to be split. The jump table of a region R_{out} contains the destination labels of all its outward *goto*s. When *ReturnFlag* is greater than 1 on f_{out}'s return, (*ReturnFlag*-2) is an index to R_{out}'s jump table Thus we can use a computed goto statement to direct the control to the proper label.

For example, the region in function *foo* contains only two outward *goto*s and both of them point to label *L*2. Thus, the jump table contains only one element *L*2. In the new function (Line 15 and 18 in Figure 4.12), the original *goto L*2 is replaced with a statement that stores $(0+2)$ into the *ReturnFlag* (0 is the index to the jump table and 2 is a constant that skips over the other two return methods). In the original program unit, we get the proper jump table index (0) by (*ReturnFlag*-2). Thus, the control is directed to the first label in the jump table: *L*2 (Line 13 and 14 in Figure 4.11).

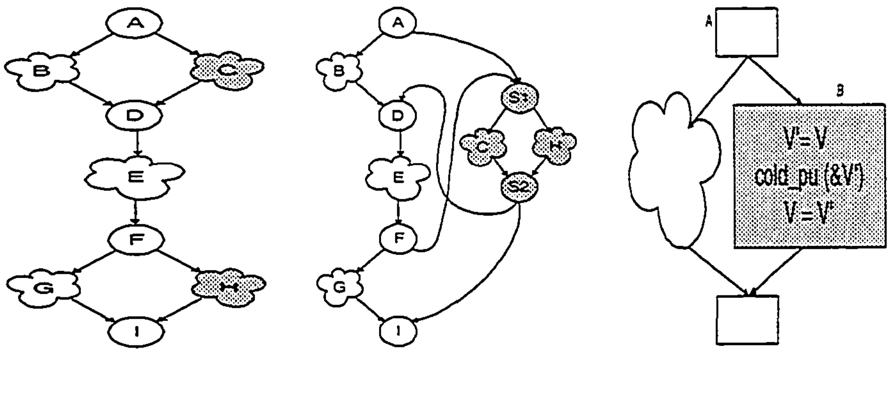
The second new symbol *ReturnValue* is used to handle regions with a *return* statement. *ReturnValue* is a variable of the return type of the original program unit. When f_{out} needs to return a value, the new function does not return the value directly to $f_{leftover}$. Instead, it saves the return value in *ReturnValue* and set the *ReturnFlag* to 1. When *ReturnFlag* is 1 on f_{out}'s return, $f_{leftover}$ should directly return the value stored in variable *ReturnValue* to f_{caller} (Line 12 in Figure 4.11).

v. **Put f_{out} into the compiler control.** The compiler compiles a single program unit at a time. After outlining, f_{out} has to be placed in the background so that the compiler can proceed with the compilation of $f_{leftover}$. ORC maintains a list of program units to be compiled. Thus, a control block of f_{out}, including its WHIRL tree and symbol table information, is inserted into this list.

4.3.5 Performance Tuning

Proper thresholds for outlining benefit and cold region size are important. After experimentation with a large set of thresholds the values of 1000 for the benefit threshold and 10 for the cold region sizes were selected. This section describes some important performance tuning, based on different strategies, for the outlining optimization.

Independent outlining vs. Collective outlining. Regions to be outlined may be scattered throughout f_{host}. Two possible outlining strategies are *independent outlining* and *collective outlining*. In independent outlining, each cold region is split into a separate function as shown in Figure 4.13(a). In collective outlining a single f_{out} function contains all outlined regions (Figure 4.13(b)). In this case the *Flag* parameter is used to dispatch control to the correct region whenever f_{out} is invoked. Each outlined region in $f_{leftover}$ is replaced with an assignment to *Flag*, a call site to f_{out} and control-flow-patching code after the call site. A drawback of collective outlining is a more complex CFG in $f_{leftover}$ which may be difficult for downstream compiler analysis.



(a) Independent (b) Collective (c) Alias Agent

Figure 4.13: Different outlining strategies (shaded code is cold)

Alias Agent When the address of a variable x is passed as a parameter to f_{out}, imprecise alias analysis will conservatively assume that x can now be aliased to any other variable that f_{out} has access to. This conservative assumption may prevent downstream optimizations and result in a serious performance penalty. For instance, variables that were kept exclusively in registers might be spilled after outlining is implemented because of imprecise alias information. This situation occurs often in ORC 2.1 and constitutes a performance hazard. Our solution is to introduce an *alias agent technique* to eliminate this serious side-effect. Each variable v whose address is passed to f_{out} has a corresponding alias agent v'. An alias agent is a new local variable introduced in f_{host}. Just before the invocation of f_{out}, the value of v is copied into v'. Then the address of v' is passed to f_{out}. Upon return from f_{out}, the value of v' is copied into v as shown in Figure 4.13(c). Both copies occur in the same cold basic block that contains an invocation to f_{out}. Without *alias agents*, we found that ORC often places memory spills in hot paths, *e.g.* into block A instead of B in Figure 4.13(c), degrading runtime performance significantly.

4.4 Results

An experimental investigation on SPEC2000int benchmarks reveals that:

- Outlining reduces the size of hot functions by up to 97% and incurs less than 0.21% increase in runtime function calls.
- Alias agent combined with independent splitting is the best outlining strategy. The alias agent is crucial to avoid performance degradation caused by memory spills in hot paths.
- Function outlining has less effect on enabling aggressive inlining than previously expected. Performance improvement from function outlining alone ranges from -0.62% to 4.1%. When partial inlining is enabled, performance increases range from -0.85% to 5.75%.

4.4.1 Experiment Configuration

Experimental results were obtained on an HP ZX6000 workstation with a 1.3GHz Itanium-2 processor, 1 GB of main memory, 32KB of L1 cache, 256KB of L2 Cache, and 1.5MB of on-die L3 cache. The operating system was Red Hat Linux 7.2 with a 2.4.18 kernel. This experimental study is based on SPEC2000 integer benchmarks.[3] All the profiling information is obtained by using standard SPEC2000 training data set and the reported runtime data is from the standard reference data set. Time was measured by the Linux *time* command and micro-architectural benchmarking is obtained with *pfmon*. All reported run-times were the average of 5 consecutive identical runs.

4.4.2 Function Outlining Performance

Figure 4.14 shows the performance changes caused by the four strategies described in Table 4.3. Combining alias agent with independent splitting, *A-I*, usually out-performs the other strategies. Noticeable performance improvements are observed on `gap`(1.1%, `gcc`(1.2%), `perlbmk`(4.1%). When alias

[3] We don't include eon because our compiler cannot compile it sucessfully.

Short	Explanation
A-I	Alias agent and independent splitting.
N-I	No alias agent and independent splitting.
A-C	Alias agent and collective splitting.
N-C	No alias agent and collective splitting.

Table 4.3: Strategy combinations

agent is not used (*N-I* and *N-C*) significant performance degradation occurs. Inspection of binaries indicates that the imprecision of the ORC 2.1 alias analysis results in many additional memory spills in hot paths. Collective splitting (*N-C* and *A-C*) also degrade performance because of the adverse effects of sharing patch code.

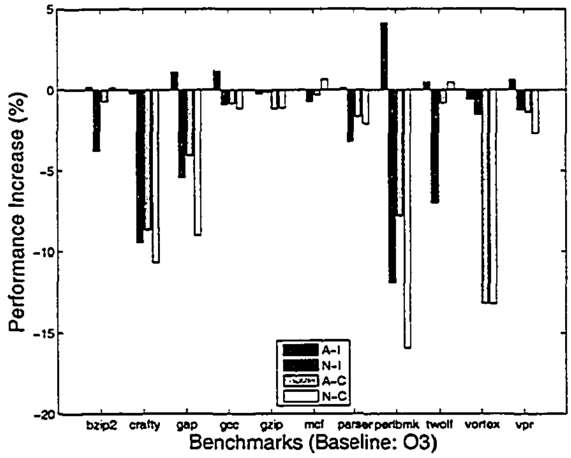


Figure 4.14: Performance of Function Outlining

4.4.3 Outlining Statistics

Table 4.4 presents some stiac statistics for *A-I*. The first row shows that function outlining occurs in many places in gap, gcc, `perlbmk` and `vortex`. Small benchmarks such as `bzip2`, `gzip` and `mcf`, have much less regions split. *If* statements are the major source of cold regions in hot functions. The *Function Size Reduction* row shows that outlining reduces function sizes drastically (up to 97%). When the patching code is larger than the split code, outlining enlarges functions. The number of parameters needed for the outlined functions ranges from 2 to 19.

We also found that function outlining increased the number of runtime function calls by at most 0.21%. This indicates that (1) the heuristics successfully avoid outlining hot regions and (2) the SPEC2000int training data set is representitive of the reference data set.

Benchmarks		bzip2	crafty	gap	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
Number of Regions Split		5	59	192	388	1	3	36	415	4	410	61
Control Flow Construct	if	5	54	186	354	1	3	36	374	3	398	61
	switch	0	5	0	12	0	0	0	33	0	7	0
	early return	0	0	6	22	0	0	0	8	1	5	0
Function Size Reduction	min (%)	-14	1	-63	-37	35	7	-5	-23	3	-35	26
	max(%)	17	1	97	84	35	15	26	89	23	71	55
	avge(%)	1.5	9.8	19.7	15.6	34.5	11.0	16.1	21.6	9.6	18.2	7.0
Number of Parameters Passed	min	2	2	2	2	2	3	2	2	3	2	2
	max	2	6	8	19	2	5	6	16	5	18	9
	average	2.0	4.8	5.5	8.5	2.0	4.3	3.7	9.8	3.8	7.3	4.1

Table 4.4: Statistics of outlining

4.4.4 Partial Inlining

A-I is also the best strategy for partial inlining. Figure 4.15 shows the performance improvements due to *A-I* partial inlining.[4] perlbmk and gap improved by 5.75% and 3.90% because of partial inlining. vpr and parser have minor performance degradation (0.86% and 0.51%).

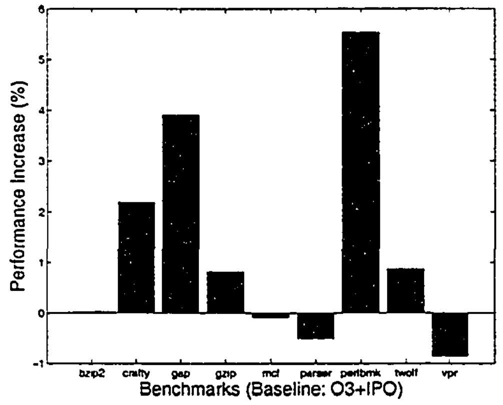


Figure 4.15: Performance of Partial Inlining

Surprisingly, very few additional call sites are inlined due to function outlining. The only benchmarks that have extra call sites inlined are: gap (10), gzip (5), parser (3), and perlbmk (5). These call sites contribute a very small percentage of the runtime function calls (less than 0.6%).

There are two major reasons that impede more aggressive inlining in the SPEC2000 integer benchmaks. More aggressive inlining is prevented by hot functions that cannot be inlined because they are too large even after outlining. To make matters worse, because few of them are leaf functions in the call graph, they often absorb other functions during function inlining and become even larger. Moreover, benchmarks that tend to benefit from partial inlining are often large benchmarks, such as perlbmk and gap, where runtime function calls are distributed among many call sites and there are no dominating call sites. Thus, a small increase in the number of inlined sites is unlikely to yield significant changes in performance.

Thus, where is the the performance improvement of partial inlining coming from? First, function outlining segregates regions with heterogeneous execution frequency into separate functions and improves code placement and cache efficiency. Second, our outlining includes better switch optimization and explicit memory spills in the cold paths (*i.e.* alias agent), which might help the compiler to do a better job in other optimizations, such as code scheduling and register allocation. Figure 4.16 shows the changes in the number of processor stalls and retired instructions when partial inlining is enabled. There is a

[4] Results for partial inlining for gcc and vortex are ommited in this version of the paper because of an under-investigation bug in the compiler.

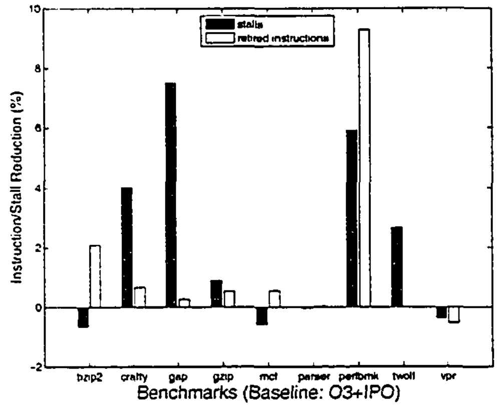


Figure 4.16: Effects on Stalls and Instructions

positive correlation between the number of retired instructions and/or processor stalls and improvements from partial inlining. For example, the processor stalls and retired instructions in perlbmk are reduced by about 5.9% and 9.3%, respectively. In other benchmarks, process stalls are significantly reduced in crafty, gap, and twolf. Also, bzip2 also show reduction in retired instructions (around 2%).

4.4.5 Aggressive Partial Inlining

Are the current inlining heuristics aggressive enough to take advantage of function outlining? We designed experiments where the compiler is forced to inline the outlined versions of hot functions without considering the function size constraints. The result is significant performance degradation, ranging from -1.3% to 9.6%. This result is disappointing but not surprising. As we mentioned in Chapter 3, the inlining heuristics in ORC are already aggressive and were carefully tuned. At the start of our outlining project, we knew that, excluding the large hot functions, the inlining heuristics had been so aggressive that a little more inlining would result in performance degradation. The experimentation with aggressive partial inlining shows that this sensitivity still holds for functions whose sizes were reduced by outlining.

4.5 Related Work

There is previous work related to several aspects of the design and implementation of the outlining presented in this thesis.

4.5.1 Function Splitting

Several researchers have studied function splitting [21, 64, 65, 72, 88, 91, 92, 93]. In their well-known code positioning paper, Pettis and Hansen split a function and put the hot and cold code far apart in the address space [72]. They don't generate new functions for the split code. Control transfer between hot and cold code is by explicit jump instructions. When the two parts are located too far away from each other, code stubs are needed for relaying jumps. Their motivation is to reduce the size of the primary function, which contains the hot code, to allow important related code to co-exist in the instruction cache or to be placed in the same memory page. Their function splitting is only intended for code placement, therefore it is implemented at the link phase. Their control flow patching method breaks the address integrity of a function and is difficult to implement and maintain in high-level optimizations.

Castelluccia *et al.* and Mosberger *et al.* use outlining to increase the code density of network protocol code [21, 64]. However, they only handle *if* statements. Our experimental work shows that *switch* statements and early returns are also important causes for unbalanced execution frequency in standard benchmarks. Thus our outlining framework includes outlining of these constructs.

Muth *et al.* proposed the implementation of partial inlining in a link-time optimizer called ALTO [65]. They generate a new program unit to hold all the split code. Once control is transfered to the program unit containing the cold code, it cannot return to the unit containing the hot code. As a consequence, the cold program unit has to clone any portion of the hot code that is reachable from the cold region, making the code bloat problem worse. While we share Muth's motivation, we think that their outlining and partial inlining occur too late in the compilation process to allow other optimizations to benefit from partial inlining. Very few optimizations occur after linking. In contrast, our outlining occurs in the very beginning of the backend. Early outlining enables aggressive inlining and potentially benefits all the later optimizations.

Way *et al.* experimented with partial inlining in early phases of a compiler backend [91, 92, 93]. Their inlining is an enabling technique to build interprocedural regions and reduce optimization costs. They made very limited exploration on partial inlining in [93] and achieved less than 1% performance improvement. The shortcoming of their work is that their function splitting is conservative because they only consider outlining when inlining a callee. Comparatively, our outlining is aggressive because, no matter a function will be inlined or not, it is always outlined whenever we think the cold regions in a hot function is large enough. We have shown that partial inlining is not effective on enabling more aggressive inlining and the major performance improvement is from the second-tier benefits of function outlining. Better performance improvement reported in this work indicates that our aggressive outlining is better than Way's approach in terms of exploiting the the second-tier benefits of function outlining.

Suganuma *et al.* propose partial inlining for a Java just-in-time compiler. Their work takes advantage of the on-stack-replacement technique [42] supported by their Java virtual machine and therefore they do not need to use parameter-passing for control flow patching during outlining, as we did in our work. Whaley *et al.* also differentiate cold and hot regions in dynamic compilation frameworks: they selectively compile hot code and leave cold code unchanged [94]. This technique reduces the compilation time. Because compilation time is counted in the runtime performance in dynamic compilation systems, partial compilation of a function alone yields impressive performance improvement. However, compilation time is irrelevant to runtime performance in static compilers. Moreover, transition from compiled hot code to interpreted cold code is only feasible in dynamic compilation systems.

4.5.2 Region Formation Algorithm

Hank's intra-procedural region formation method [38] is a generalization of the IMPACT compiler runtime feedback-based trace selection algorithm [70]. Hank analyzes the CFG of a function to identify a hot region that includes the entry and exit of the function. Using the most frequent basic block as a seed for the hot region, Hank's algorithm first traverses upward and downward in the CFG to find a most desirable path as the seed path. During this seed path generation process, Hank determines the desirability according to the frequency relevance of the successor or predecessor basic blocks. Once the seed path is selected, the algorithm tries to use similar frequency heuristics to include more relevant basic blocks to generate the hot region. Hank's region formation occurs after aggressive inlining. Very large functions generated by aggressive inlining may significantly increase compilation time. Hank's motivation is to repartition a large function into small regions to control the compilation time while exposing important optimization opportunities. Way *et al.* later contended that aggressive inlining itself might be expensive. Instead they propose an extension to Hank's algorithm that integrates inlining with inter-procedural region formation [91, 92].

The CFG-based region formation of Suganuma *et al.* tries to identify cold regions in a hot function [88]. They first select some seed basic blocks as *rare* or *non-rare* according to some pre-defined heuristic. Then this information is propagated along backward data flow until it converges. Then the CFG is traversed again to decide the regions and the transitions between them.

All these region formation methods, including ours, try to separate code segments with heterogeneous execution frequency. We take advantage of frequency-annotated high-level intermediate representation to implement our region formation. Our transformation is closer to the source code and done without CFG formation. We claim that our implementation is more straightforward and easier to debug. When CFG is used to form regions high-level control flow information is lost. For instance, the absence of this information might lead to collective outlining, which we found to be not efficient in Section 4.4.

4.5.3 Preservation of Semantics in Splitting

Komondoor *et al.* use function splitting to abstract repetitive code segments to a new function so that the program becomes easier to understand and maintain [57, 58]. We have a different goal: to separate the cold code from a hot function. However, their semantics-preserving methods handle problems that are similar to the ones that we met in our study: some statements, known as *exiting jumps* in their work, such as *returns* in the outlined region and *goto*s from the outlined region to the leftover region should simply not be included in the outlined function. Their splitting candidates are limited to single-entry regions while our splitting framework can handle side-entries to a region. Thus our control flow patching work can be seen as a superset of their techniques.

4.5.4 Code Layout

Besides reducing function size thus enabling more aggressive inlining, function outlining has another important advantage: it improves code layout. Our study reveals that better code layout is actually the major benefit of function outlining. There has been a lot of code layout work in the literature. Some of these studyies only conduct code layout at a function granularity [35, 39]. Other research works do basic block layout [72]. Some of these works tried to separate cold code segments from the hot ones in the same function [72]. But they did this in an architecture-specific way (see section 4.5.1). On the contrary, our function outlining is architecture-neutral and can be implemented in early phases in a compiler.

Chapter 5

Data Outlining or Reshaping

5.1 Introduction

Fast advances in semiconductor fabrication, architectural innovation and exploitation of instruction-level parallelism (ILP) ensures that the performance potential of modern processors continues to increase at a substantial speed. However, the performance of a computer system is not solely determined by the processor. To maintain a high utilization of its functional units, a fast processor must be efficiently fed with instructions and data by a memory subsystem. Unfortunately, the speed of memory continues to lag behind that of processors. In recent decades, the clock rate of processors has approximately doubled every three years while the DRAM access speed only increased about 50% [60].

The solution to overcoming this increasingly insurmountable memory wall is to improve caching systems. Because of fabrication constraints, power consumption, and economics, the cache size is often very limited when compared with main memory. Hence, efficient utilization of cache is crucial for performance. However, because cache is transparent to application programmers, programs are often written without taking cache efficiency into account. The typical programmer designs data structures in a semantic-oriented fashion. Such structures are usually easy to read and understand, but often lead to suboptimal performance at run time. A compiler can analyze the memory reference pattern of a program and devise a new layout that increases the locality of references and thus improves the efficiency of the memory system. Similar to traditional code transformations, these data transformations must be transparent to the programmer, performed automatically, and safe.

This chapter describes *Forma*,[1] a framework that improves the data cache efficiency of arrays of aggregate data structures. An extension of this framework to handle linked data structures (LDS) is discussed in Section 6.2.2. Aggregate data types, such as `structs` and `classes`, are used to model objects in imperative programming languages. When the object modeled has many fea-

[1] *Forma* is a Latin, and Portuguese, word for "shape".

tures, the aggregate data type becomes very large. Arrays that contain large data structures with many fields occur frequently in contemporary programs. Because programmers often arrange the fields in a data type in a way that is semantically meaningful, there is a tension between software engineering and performance engineering that presents itself in two ways. First, the frequency of access to fields in the same data type may vary significantly, with *hot fields* accessed very frequently and *cold fields* seldom referenced. Placing fields with very different access frequencies together in memory hurts performance because the cold fields pollute the data cache and waste memory bandwidth. Second, the runtime data access pattern might not be consistent with the access frequency distribution. In other words, hot fields are not necessarily accessed together. The gap between software engineering and performance engineering can be bridged by reshaping the data at compile time. The compiler can split a large data structure into two or more smaller ones that better capture data locality. Correspondingly, an array of large data structures can be partitioned into two or more arrays of smaller data structures. For iterations that only manipulate certain fields of the array, data reshaping can significantly improve data locality and reduce the memory footprint, resulting in better data cache efficiency.

The main contributions of this chapter are:

- *Forma*, a practical data reshaping framework that can be used to automatically analyze and transform real C/C++ programs. *Forma* consists of a data shape analysis, including both alias analysis and data type analysis, structure partition planning and array reshaping transformation. *Forma* has been integrated into the IBM® XL C/C++ V7.0 compiler.

- A set of safety-checking rules to ensure that the compiler's data reshaping plan is safe in programs that are written in type-unsafe languages such as C and C++.

- An empirical study of two orthogonal reshaping decisions: frequency-based object partition × affinity-based object partition × maximum object partition; and address-arithmetic-based × pointer-based array splitting. Some important but subtle insights on data reshaping are exposed by a thorough analysis and empirical study.

The rest of the chapter is organized as follows. Section 5.2 introduces *Forma* and two design dimensions of data reshaping. Then the performance of the different data reshaping approaches is studied in Section 5.3. Section 5.4 discusses related work on data cache optimization.

5.2 Data Reshaping

5.2.1 Overview

Previous research work on field placement and data structure splitting appears in [36, 71, 73, 101]. However, some of these studies use error-prone human inspection of C applications to make sure that the transformations are safe [36, 101]. Applying a type-safe-oriented optimization to a type-unsafe language without a proper safety assurance mechanism is unacceptable in production compilers. Rabbah *et al.* use field-insensitive Steensgaard style analysis to find the alias sets that need to be updated upon data splitting [71, 73]. However, alias analysis alone cannot guarantee the transformation safety. Also, as our work illustrates, field-sensitivity in alias analysis is crucial to uncovering important array reshaping opportunities in many integer benchmarks.

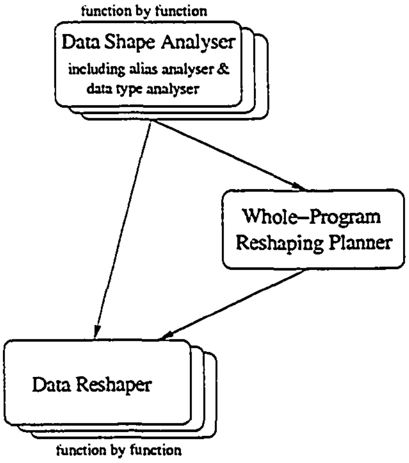

function by function
Data Shape Analyser
including alias analyser & data type analyser
Whole–Program Reshaping Planner
Data Reshaper
function by function

Figure 5.1: The *Forma* data reshaping framework

This chapter describes *Forma*, a complete framework that performs ***automatic*** and ***safe*** data reshaping on type-unsafe programming languages such as C/C++. In contrast with existing work, *Forma* is fully automatic, safety-guaranteed, and more aggressive on alias analysis in terms of field sensitivity than previous work. *Forma* was designed and implemented in the IBM®XL C/C++ V7.0 compiler suite.

As illustrated in Figure 5.1, *Forma* consists of three components: a data shape analyzer, a structure partition planner, and a whole-program data reshaper. *Forma* requires two passes through the entire program: a data shape analysis pass and a data reshaping pass. The data shape analysis pass includes alias analysis and data type analysis. In this pass, a storage shape graph [33] is constructed to model the aliasing relationships in the program. *Forma* also examines whether the data types[2] of the members in an alias set are consis-

[2] In this research, we use *data type*, *data shape* and *data view* interchangeably.

tent throughout the entire program. If array reshaping is deemed safe and beneficial, at the end of the first pass *Forma* creates a partition plan for the composite data structure. Then the data reshaping pass adjusts the memory accesses according to this plan.

5.2.2 Data Shape Analysis

An inter-procedural data shape analysis generates information about alias relationships and data shape consistency. Alias information provides a conservative approximation of sets of data objects that potentially reside in the same memory location. Data shape includes *structural shape* and *array shape.* Structural shape describes the field-level view of a singular data object, *i.e.* the number of byte-level fields, the offset and size of each byte-level field. Array shape is the view of an array. It consists of the number of dimensions and the stride for each dimension of the array. If the view of an array throughout the program is not consistent, the compiler must be conservative and give up data reshaping optimization on the array.

Inter-procedural Alias Analysis

In a program written in a pointer-rich language, such as C and C++, reshaping a data object might impact the whole program because of aliasing relationships. Therefore, a compiler needs to modify all the affected references when it reshapes a data object. *Forma* focuses on reshaping arrays. If an array is to be reshaped, all the references to the array area need to be modified accordingly. This comprehensive transformation requires *Forma* to conduct an inter-procedural alias analysis to collect all the pointers pointing to the array area.

```
char *pc;  int i,j;
struct A {char *f1; struct A *f2;} *p1, *p2;
i = j = 10;
p1 = malloc (1000 * sizeof (A));
p2 = &(p1[j]);
p1[j].f1 = pc;
p1[i-1].f2 = &(p1[i]);
// pc = (char*) p1;
```

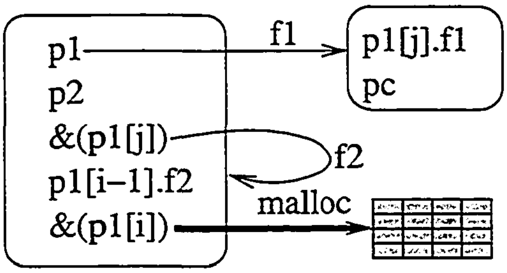


Figure 5.2: Field-sensitive Steensgaard alias analysis

Inter-procedural alias analysis techniques differ in flow sensitivity, context sensitivity, field sensitivity, and so on. A good survey of these techniques can be found in [77]. *Forma* implements a Steensgaard style alias analysis [83] that has the following characteristics:

- It is flow-insensitive. It conservatively assumes that all the statements in a procedure will be executed in arbitrary order. Thus the control flow information is irrelevant.

- It is context-insensitive. It does not differentiate the alias relationship created in different calling contexts.

- It is unification-based. Whenever a pointer assignment is met in the analysis, the alias sets represented by the right-hand side pointer and by the left-hand side pointer are merged. The points-to sets of these pointers are also merged. This assumption eliminates the iteration over the control flow graph (CFG) that is required by inclusion-based alias analysis. Therefore, the analysis can be completed with a single pass through the entire program.

Steensgaard's alias analysis was selected for *Forma* for two reasons. First, it is much simpler and faster than other alias analysis [43]. Second, the precision of alias relationships provided by Steensgaard's alias analysis is sufficient for data reshaping. To reshape an array, it is sufficient to know all the data accesses to the array. There is no need to know precise aliasing relationships such as whether two pointers actually point to exactly the same element of the array. The extra precision provided by flow-sensitivity, context-sensitivity, and directionality in a more complex and more expensive alias analysis would be redundant.

Field-sensitivity is important for data reshaping because large data structures, which are amenable to data reshaping, often contain pointers pointing to different data types. A field-sensitive alias analysis is necessary to distinguish the alias relationships among different fields or between a field and its *host object*. The host object is the object that contains the field. For instance, Figure 5.2 depicts a code segment and the corresponding storage shape graph generated by a field-sensitive Steensgaard alias analyzer. The pointer manipulation in the example is pretty common in practice. The bold arrow represents the allocation site in the program. Thin arrows represent the points-to relationships among expressions. In the example, the points-to set is divided into two categories according to the fields. That is, field `f1` and field `f2` should never reference the same address. The field-sensitivity is crucial for this code segment. In a field-insensitive analysis, access to any field is regarded as an access to the entire host object. Therefore, pc and `p1[j].f1` would be in the same alias set as `p1`. *Forma* would conclude that the `p1`'s alias set could be either an `A`-typed pointer or a `char` pointer. Then *Forma* would have to give up the data reshaping of p1's alias set because the data types of the members in the alias set would be regarded as inconsistent. In *Forma*, field-sensitivity is achieved by a technique similar to those in [82, 98].

Reshaping Safety

C/C++ are weakly type-checked programming languages. This means that even extensive type checking in a compiler front-end cannot detect all unsafe operations. Some of the type loopholes were intentionally included in these languages to enable performance-efficient and code-convenient implementations of system software. For example, let's examine the commented statement in Figure 5.2. The address of the allocated memory block is cast to type *char* and assigned to pointer pc. This casting is very common in the implementation of low-level communication libraries: a buffer is filled with an array of high-level data objects and then streamlined and sent to the lower transferring layers. Reshaping on data that has incompatible types is dangerous and is strictly avoided in our work.

In *Forma* two intrinsic data types are compatible if their sizes are identical. This is a more relaxed definition when compared with that from [51]. Two aggregated data structures are compatible if (1) they have the same number of byte-level fields,[3] (2) corresponding fields have the same offset and size, and (3) their addresses are either identical or don't overlap with each other. Two arrays have compatible types if (1) their element types are compatible, (2) they have the same dimensions, and (3) the element sizes of corresponding dimensions are also identical. Two pointers are of compatible types if and only if the data they point to have compatible types.

Forma conducts type-compatibility checks to avoid dangerous data reshaping. *Forma* does not attempt to improve the type-safety of a program. The data reshaping transformation must be carefully implemented to avoid introducing new, potentially unsafe, runtime type errors to the program. Safety is achieved by integrating a type compatibility analysis with the inter-procedural alias analysis: the types of the members in an alias set must be compatible throughout the application. The inter-procedural alias analysis keeps track of the types of each alias set. Once a type incompatibility is found, the alias set is abandoned for reshaping analysis.

The compatibility rule is necessary to ensure the safety of the transformation. It requires that all the access patterns in an alias set be verifiably consistent. For example, if a pointer is passed to system libraries and the compiler cannot examine the access pattern in the libraries, the alias set represented by the pointer must be abandoned. Fortunately, a large set of programs satisfy these seemingly restrictive conditions [23, 67].

Section 5.2.4, introduces two array splitting strategies. One of these strategies, address-arithmetic-based splitting, requires an extra restriction to ensure

[3] We assume the bit fields are converted to byte-level fields.

its safety.

5.2.3 Structure Partition Plan and Array Reshaping

If an array is deemed safe for reshaping, *Forma* makes a partition plan for the aggregated data structure of the elements of the array. The shape of the original array will change to satisfy the data structure partition. Then *Forma* transforms all the accesses to the array to make them compatible with the new array shape and the new aggregated data structures.

This section describes three structure partition planners. Two approaches to reshape an array according to the structure partition plan are discussed in Section 5.2.4. Section 5.3 presents a detailed empirical performance study of reshaping and splitting.

Structure Partition Planner

A structure partition plan determines how the fields in the original data structure should be reorganized into new data structures. For example, consider the four-field data structure, O_{orig}, in Figure 5.3. The fields of O_{orig} are numbered from F0 to F3. Each rectangle represents a field, and rectangles with the same filling pattern are fields that are always accessed together. In this example, field F0 and field F3 have high access affinity, while field F2 is always accessed alone. Assume that field F1 is very cold and the other fields are hot. In Figure 5.3, the fields in O_{orig} are reordered and split into three smaller structures: O_{base}, O_{sat_1} and O_{sat_2}. Each new structure has its own size and offset in the reorganized data structure. O_{base} is the *base structure* or *base object* and starts from offset 0. The other new data structures, O_{sat_1} and O_{sat_2}, are the *satellite structures* and are placed immediately after O_{base}.

Array reshaping is based on a data structure partition plan. After array reshaping, different partitions of the same object might be placed far apart from each other. Figure 5.4(a) shows an array, A_{orig}, of four elements of type O_{orig}. Figures 5.4(b), 5.4(c), and 5.4(d) show the effects of different structure partition plans on A_{orig}. In these figures, each rectangle is a field of O_{orig} and Fij represents the jth field in the ith element of the original array. From the array point of view, the original array A_{orig} is split into one base array A_{base}, which holds the base objects, and one or more satellite arrays. For instance, each element might be split into three new objects according to the partition plan shown in Figure 5.3. Correspondingly, A_{orig} might be split into three arrays, as shown in Figure 5.4(b). The first two rows in Figure 5.4(b) are the new base array A_{base} and the other two rows are the new satellite arrays. How the satellite fields are accessed depends on the array-splitting approach, as described in Section 5.2.4. To correctly reference satellite fields that are placed in satellite objects, extra address calculations are needed. Therefore, if hot fields are split into satellite objects, there might be a substantial increase in the number of instructions executed to compute the address of hot satellite fields.

Forma implements the following three data structure partition strategies.

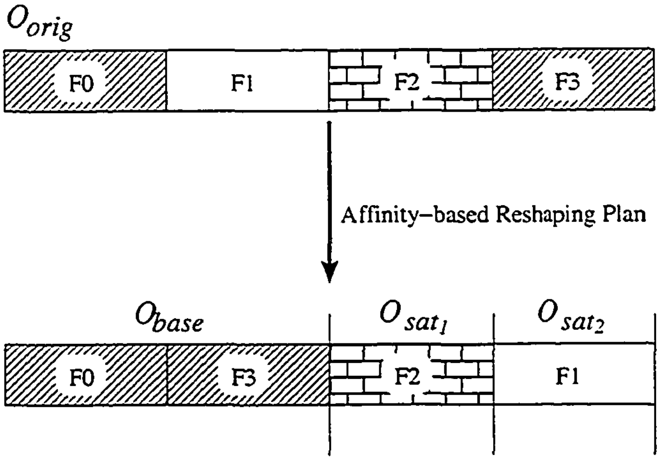
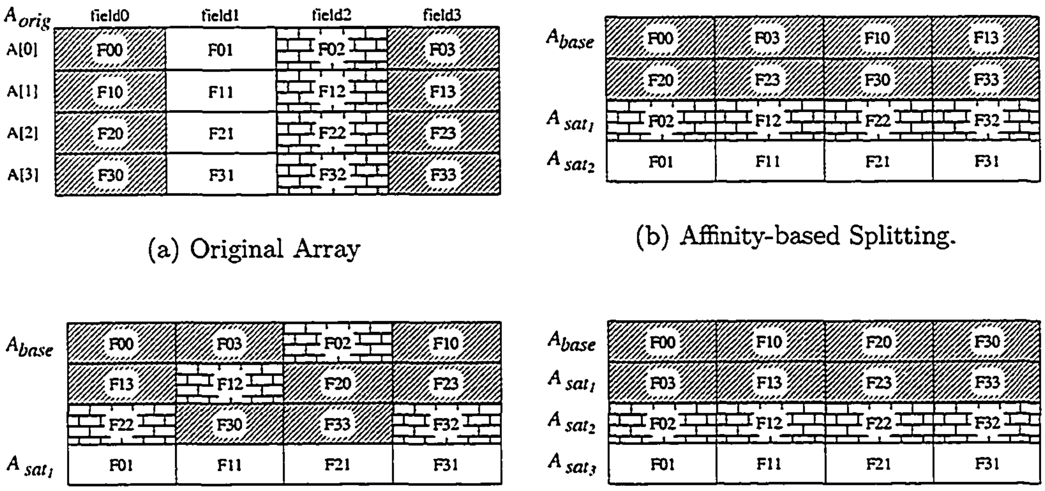


Figure 5.3: Reshaping planning (affinity-based)



(a) Original Array

(b) Affinity-based Splitting.

(c) Frequency-based Splitting.

(d) Maximal Splitting.

Figure 5.4: Different reshaping planning strategies

Affinity-Based Planner (ABP). Some of the hot fields in a data structure have higher *access affinity* than others, *i.e.*, they tend to be referenced together. Therefore, a natural solution is to split a data structure according to access affinity: the fields in a data structure are clustered by reference affinity and each group of fields is split into an independent data structure. In Figure 5.3, the original data structure is split into three new structures. The first structure contains fields F0 and F3. The second and third structures hold fields F2 and F1, respectively. Guided by the reshaping plan, a four-element array is split into three smaller arrays (Figure 5.4(b)).

ABP captures the runtime reference locality more closely than the other two planners discussed in this chapter. Frequently, however, it is not easy to find clear-cut affinity relationships like the ones in Figure 5.3. For example, a field A might have high reference affinity with two other fields B and C in different iterations, but B and C might never be accessed together. In such cases, it is difficult to find an optimal partition that respects competing affinities. The second drawback of affinity-based splitting is that hot fields without reference affinity may be placed into separate data structures and arrays. Placing hot fields in satellite arrays results in substantial address computation overhead.

Frequency-Based Planner (FBP) or *hot-cold planner* uses runtime feedback information to partition a data structure into two. The first array contains hot fields and the second array contains cold elements. For instance, the first data structure in Figure 5.4(c) contains fields F0, F2 and F3. The second data structure contains field F1. A frequency-based planner only needs to calculate addresses of satellite fields that are infrequently accessed. Therefore, FBP does not require the execution of many additional instructions. However, it neither captures the reference affinity as well as ABP does nor reduces the memory footprint as aggressively as MSP, described below, does. Moreover, FBP might waste memory bandwidth and pollute the data cache. For instance, in the example of Figure 5.4(c), when the program iterates through the fields F1, the fields F0 and F3 are uselessly fetched into cache.

Currently *Forma* uses 95% as a frequency-based planner threshold. That is, it retains the most frequently referenced fields that account for 95% of the accesses to the data structure in the base object. All other fields are treated as cold and are split to a satellite object.

Maximal Splitting Planner (MSP) splits each field of a data structure into a separate new data structure, as shown in Figure 5.4(d). After splitting, each data structure field is stored in an independent array. An obvious advantage of MSP is that it does not require profiling information. MSP ignores the reference affinity among the fields in the same data structure and seems to be too simple to be good for performance. Surprisingly, as shown in Section 5.3, MSP achieves the best or close-to-best runtime performance among the different reshaping planners studied in this chapter. This is because MSP has three important but subtle advantages.

First, MSP always achieves the smallest memory footprint for stride-1 iterations on the array. This is because each field has its own array and no irrelevant data is fetched into cache. In contrast, neither ABP nor FBP can guarantee that the partition respects access affinities for all traversals of the arrays. Therefore, even when *Forma* attempts to take into account affinity or frequency information, there may be traversals that fetch irrelevant fields into the cache. In modern processors, where the latency to bring data from memory is high, the smaller memory footprint generated by MSP can be a decisive advantage. For instance, it can significantly reduce misses in the lower levels of the memory hierarchy as well as in the translation look-aside buffer (TLB).

Second, MSP is especially suitable for processors that feature a hardware prefetching mechanism, such as the IBM POWER4$^{\text{TM}}$ processor [46]. A stream is a sequence of memory loads that access two or more contiguous data cache lines in either ascending or descending order. The processor monitors cache misses closely. Once misses on two consecutive cache lines are detected, a directed stream prefetching is triggered, and data from the memory area in the directed stream is fetched into higher levels of the memory hierarchy. The combination of hardware stream prefetching with high cache associativity allows independent streams that are accessed together to be simultaneously prefetched into different cache areas. This simultaneous prefetching tends to compensate for the loss of field affinity in MSP. Because the stride on each split array is smaller than those in arrays organized according to affinity, stream prefetching should work more efficiently. Therefore, there is no need to worry about the field affinity because the prefetching mechanism covers multiple streams consisting of fields with affinity. Fortunately, each processor supports eight independent streams, which seems to be sufficient for most applications. In the entire SPEC2000 benchmark suite, we haven't encountered any important array traversal that accesses more than eight fields. Moreover, even when there are more than eight streams in a loop, the XL compiler is able to distribute them into several smaller loops through loop fission [96].

Third, because maximal reshaping converts each field into a single object, the host object of a field contains only the field itself. Therefore, the address of the host object and the address of the field are the same and there is no need to compute the field offset. As a consequence, the address calculation for satellite fields is simpler when maximal reshaping is used.

MSP sacrifices field affinity to take advantage of field locality and reduce memory footprint. Loss of field affinity is compensated for by hardware stream prefetching and by higher associativity in modern cache systems.

A drawback of MSP is similar to the drawback of the affinity-based planner: all the fields, except the field in O_{base}, need to be accessed indirectly. Thus, if the field in O_{base} does not dominate the access frequency of the data structure, many additional instructions are executed to calculate the address of satellite objects.

5.2.4 Array Reshaping

The last phase of array reshaping transforms the program according to the reshaping plan. To apply array reshaping to an alias set, the allocation site and all related data accesses through pointers in the alias set should be transformed to reflect the change of the data view.

The design of *Forma* considered two transformation approaches: address-arithmetic-based splitting and pointer-based splitting. In address-arithmetic-based splitting, no extra fields are introduced during the transformation, and the address of a satellite field is calculated from its corresponding base object. Examples include the transformations shown in Figure 5.4(b), 5.4(c), and 5.4(d). In pointer-based splitting, extra field pointers are introduced in the base object to link each satellite object to its base object. Soon after the array is allocated, these extra field pointers are initialized to point to the corresponding satellite objects. Therefore, accesses to the satellite fields are transformed to accesses through the extra pointer dereferences.

Rule	Original	After address-arithmetic-based transformation
1	(O_{orig} *) p	(O_{base} *) p'
2	&(A[k])	&(A_{base}[k])
3	&(p -> basefield)	&(p' -> basefield')
4	&(A[k].basefield)	&(A_{base}[k].basefield')
5	&(p -> satfield)	index = (p'-A_{base})/LEN(O_{base}) &(A_{sat_i}[index].satfield')
6	&(A[k].satfield)	&(A_{sat_i}[k].satfield')
7	allocation site: A=new(N*E)	A_{base}=new(N*E) for i ∈ [1, SatNum] A_{sat_i} = A_{base}+ offset$_i$*N

Table 5.1: Address-arithmetic-based reshaping

Rule	Original	After pointer-based transformation
1	(O_{orig} *) p	(O_{base} *) p'
2	&(A[k])	&(A_{base}[k])
3	&(p -> basefield)	&(p' -> basefield')
4	&(A[k].basefield)	&(A_{base}[k].basefield')
5	&(p -> satfield)	&(p' -> pointer$_i$ -> satfield')
6	&(A[k].satfield)	&(A_{base}[k].pointer$_i$ → satfield')

7	allocation site:
	A=new(N*E)
		A_{base}=new(N*NewE)
		for i ∈ [1, SatNum]
		for j ∈ [0, N-1]
		A_{base}[j].pointer$_i$ = &(A_{sat_i}[j])

Table 5.2: Pointer-based reshaping

Tables 5.1 and 5.2 compares the differences between address-arithmetic-based reshaping and pointer-based reshaping. In the table, p is a pointer to an element in the array; `A` is the array's base address; `basefield` and `satfield` represent fields falling in O_{base} and O_{sat} in the plan, respectively. `SatNum` is the number of satellite arrays. `N` is the number of elements in the array. `E` is the size of each element in the original array and `NewE` is the size of the original element plus the sizes of the pointer fields introduced in a base object. The primed versions (such as `p'` and `satfield'`) represent their counterparts after reshaping. Tables 5.1 and 5.2 presents seven rules to transform references into reshaped objects and arrays.

- The 1st and 2nd rules say that, after reshaping, the role of the original object is taken by the base object. A pointer `p` that pointed to O_{orig} before reshaping is replaced by a pointer `p'` to O_{base} after reshaping. All the element-wise pointer manipulations are transformed to manipulate the base object. For example, p++ means `p = p + sizeof(`O_{orig}`)` in the original program. It should be transformed to `p' = p' + sizeof(`O_{base}`)` after array reshaping.

- The transformation for the base fields (rules 3 and 4) is straightforward: their addresses are acquired by applying their new offsets in O_{base} to the pointer to O_{base}.

- The address calculation for satellite fields (rules 5 and 6) differs between the two approaches. In the pointer-based approach, the satellite fields are accessed via newly introduced pointer fields for the corresponding satellite object. In the address-arithmetic-based approach, the index of a satellite object equals $\frac{p' - A_{base}}{\mathrm{LEN}(O_{base})}$, where `LEN` is the size, in bytes, of O_{base}. This index is then used to access the corresponding element in the satellite array.

- The allocation site also has to be handled differently for the two approaches (rule 7). For address-arithmetic-based reshaping, one base pointer for each satellite array is introduced, and these base pointers are initialized after the array is allocated. In the pointer-based strategy, the program enumerates each element in the base array and initializes the pointer fields for the satellite objects.

Besides the type compatibility safety check, address-arithmetic-based splitting has another restriction to avoid unsafe transformations: single-instantiation. Single-instantiation restriction says that the entire alias set is instantiated by a single allocation site in the program and that the allocation site is executed no more than once at run time. This extra restriction makes sure that the base address of the array is a constant at run time.

Both address-arithmetic-based and pointer-based reshaping have their advantages and disadvantages. Pointer-based reshaping requires fewer address calculations and does not have the single-instantiation restriction. But it has two serious drawbacks. First, it requires extra pointer fields in each base object. When the reshaping plan splits O_{orig} into several new data structures, many extra fields are required. This additional data may offset the array reshaping effort. If the plan is frequency-based splitting, only one extra pointer field is needed. Second, each access to satellite fields requires one extra pointer dereference, which is often very expensive in today's register-centered processors.

In contrast, address-arithmetic-based reshaping only requires extra address calculations when the satellite fields are accessed via an element-wise pointer (rule 5). Therefore, if individual fields are accessed often, many additional address calculations would occur. However, this problem can be mitigated by traditional optimizations such as constant propagation, common subexpression elimination, and promotion of loop invariant expressions. For example, the address of $A_{sat_i}[\frac{p'-A_{base}}{\texttt{LEN}(O_{base})}]$ is computed by the expression $A_{sat_i} + \frac{p'-A_{base}}{\texttt{LEN}(O_{base})} * \texttt{LEN}(O_{sat_i})$. At compile time, once reshaping is completed, $K = \frac{\texttt{LEN}(O_{sat_i})}{\texttt{LEN}(O_{base})}$ is constant. Consider the following extreme case that highlights the optimization potential for address-arithmetic-based reshaping: under maximal reshaping it is likely that $\texttt{LEN}(O_{sat_i})$ equals $\texttt{LEN}(O_{base})$ because both the O_{base} and O_{sat_i} contain a single field, and thus $K = 1$. In this case, `&(p -> satfield)`, that equals `p + OFFSET(satfield)`, is transformed to $A_{sat_i} + (p' - A_{base}) = p' + (A_{sat_i} - A_{base})$. The expression $(A_{sat_i} - A_{base})$ only needs to be computed once at the allocation site. In this extreme, but not infrequent, case the same number of operations are needed before and after reshaping!

Additionally, in address-arithmetic-based reshaping, the array size is exactly the same as the one before the transformation. Maintaining the same memory requirement makes the transformation safer than the pointer-based reshaping. The extra memory necessary for pointer fields may cause the program to fail when there is not enough free memory.

Although the address-arithmetic-based reshaping strategy requires single instantiation, this restriction is not a serious problem for array-centered applications. It may become a problem in applications that use linked data structures (LDS). Section 6.2.2 discusses analysis and transformations that will be required to deal with the multiple-instantiation problem in applications that use LDS extensively.

With all these factors taken into account, *Forma* favors the use of the address-arithmetic-based reshaping strategy.

5.3 Experimental Study

This section presents a performance study of array reshaping. The results of this investigation may be summarized as follows:

- Data reshaping improves data-intensive programs dramatically. This study found impressive performance gains — up to 2.1 times speedup — on three benchmarks. The seemingly naive maximal reshaping achieves best or close-to-best performance improvement on the studied benchmarks.

- Data reshaping degrades the `perimeter` benchmark. A micro-architectural performance study reveals that in this benchmark additional instructions required for address calculation offset the small improvement on cache efficiency achieved by array reshaping.

- Maximal reshaping is best suited for programs with stride-1 iterations in architectures with hardware stream prefetching such as the IBM POWER™ family.

5.3.1 Experimental Platform

Forma is implemented in the IBM XL C/C++ V7.0 compiler. Thanks to the modular structure of *Forma*, it is easy to switch on different reshaping strategies and examine their effects. Table 5.3 shows the characteristics of the machines used in this performance study.

Three of these machines use processors from the IBM POWER family. The compiler is a development version of the IBM XL C/C++ that includes *Forma*. To investigate the portability of the results obtained with *Forma* in the XL compilers, the benchmarks were modified, by hand, and run on an Intel® Itanium-II machine. This hand modification consisted of inspecting the reshaping plans created by *Forma* and mimicking them in the benchmark's source codes. These benchmarks were then compiled with the Open Research Compiler 2.1 at the O3 optimization level with inter-procedural optimization. Although not applicable to a large number of benchmarks, this effort produced data that should convince developers of other compilers to consider when implementing array reshaping.

CPU	GHz	L1D, L2D, L3D, Mem	OS	Page Size
G5™	2.0	32K, 512K, 0 , 1G	Darwin® 7.5	4KB
POWER4™	1.1	32K, 1.44M†, 32M †‡ , 32G	AIX® 5.2	4KB
POWER5™	1.65	32K, 1.92M†, 36M†‡ , 16G	AIX® 5.3	4KB
Itanium-II™	1.3	16K, 256K †, 1.5M† , 1G	Linux® 2.4.18	16KB

Table 5.3: Characteristics of the experimental platforms, memory and page sizes given in bytes (†: DCache+ICache, ‡: off-die)

There is a limited number of benchmarks that are affected by array reshaping in the standard benchmark suites. This study uses two benchmarks from the SPEC2K suite: `art` and `mcf`. Standard training data is used for profiling

Label	Partition Planner	Reshaping Method
baseline	—	—
affinity	affinity-based planner	address-arithmetic-based reshaping
freq-a	frequency-based planner	address-arithmetic-based reshaping
freq-p	frequency-based planner	pointer-based reshaping
max	maximal planner	address-arithmetic-based reshaping

Table 5.4: Compiler versions in the performance study

and the standard reference data for final runtime benchmarking. The memory footprints in the final run are 4.7MB for `art` and 190 MB for `mcf`. The other two benchmarks in this study, `tsp` and `perimeter`, are from the array version of OLDEN 1.3 benchmark suite. The profiling inputs used are 10^5 for `tsp` and 10 for `perimeter`. The inputs for the final runtime measurement are 4×10^6 for `tsp` and 11 for `perimeter`. The memory footprints for these final runs are 225MB and 256 MB, respectively.

Forma implements three partition planners and two reshaping methods. Table 5.4 lists the versions of the compiler that are included in this performance study along with the label identifying each version in all the graphs. Pointer-based reshaping requires many extra pointer fields in O_{base} when O_{orig} is broken into many satellite objects. These additional pointers seriously offset the benefit of reshaping. Therefore, pointer-based reshaping is not included in the performance study of affinity-based splitting and maximal splitting.

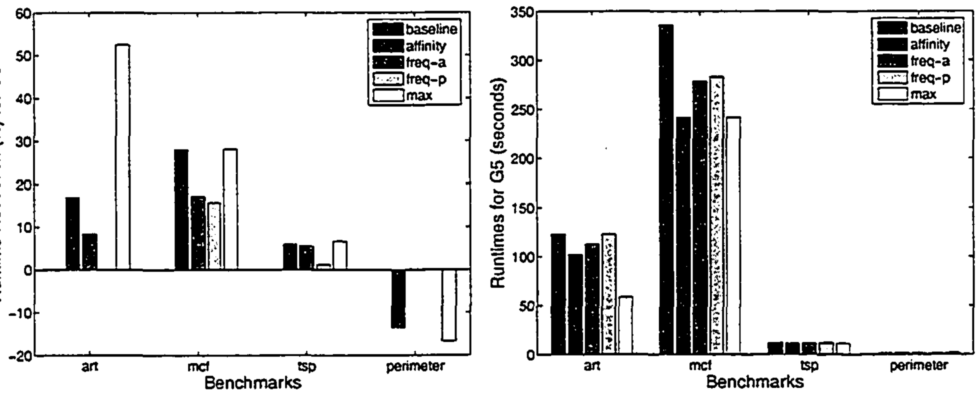


Figure 5.5: Run times on a G5

5.3.2 Run Time Improvement

Figures 5.5-5.8 presents the runtime variations among the array reshaping versions implemented in *Forma* on four hardware platforms. For each machine, the right graph presents the actual run times, and the left graph is the normalized runtime percent variation. The baseline compiler is a development

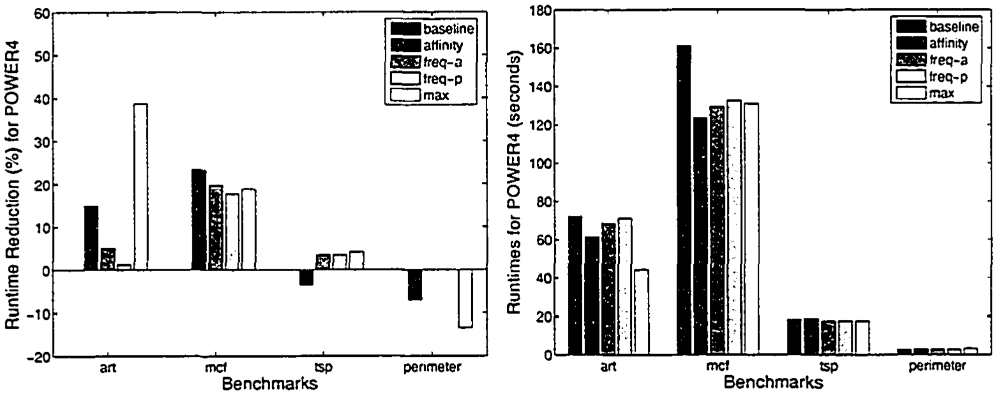


Figure 5.6: Run times on a POWER4

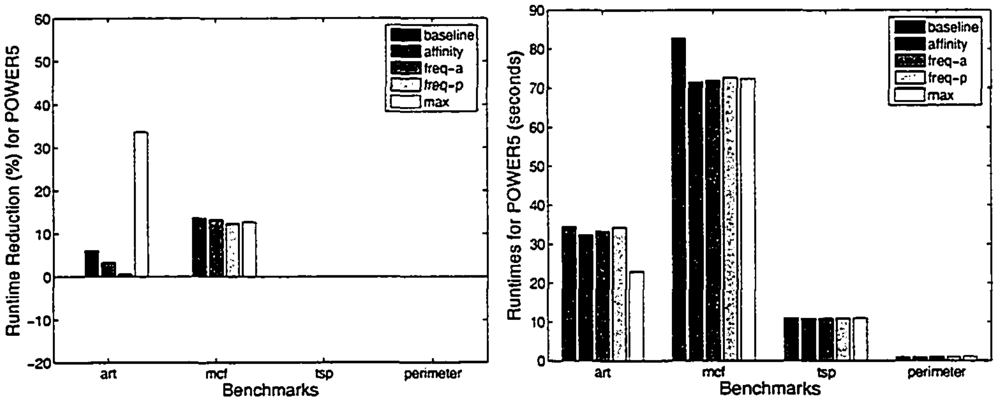


Figure 5.7: Run times on a POWER5

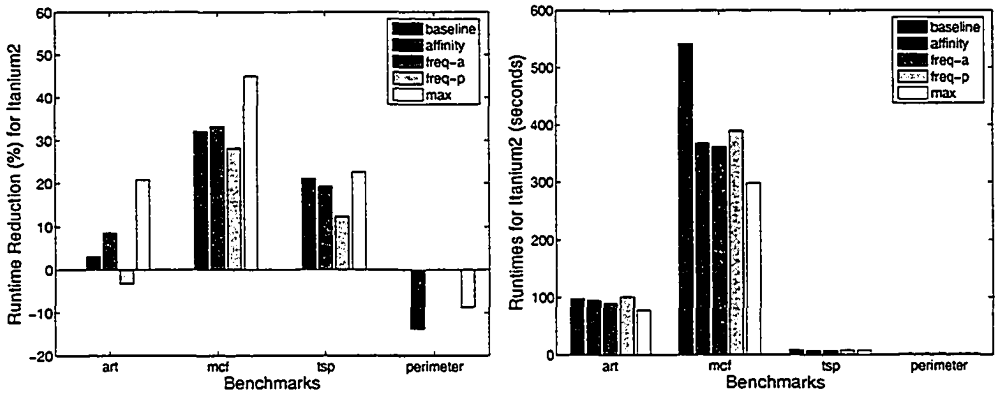


Figure 5.8: Run times on an Itanium II

version of the industry-strong XL optimizing compiler running at optimization level O5 without any data reshaping optimization. For the Itanium, the baseline is the Open Research Compiler (ORC) 2.1 at O3 optimization level with inter-procedural optimizations enabled. Array reshaping results in impressive performance improvement for `art`, `mcf`, and `tsp`.

On the G5 system, the maximal split version of `art` runs more than twice as fast as the baseline. In comparison, affinity-based reshaping improves `art` by about 17% and freq-a improves it by about 8.4%. The improvement by freq-p is negligible mainly because `art` has only one cold field, and the introduced pointer field completely nullifies the benefit of splitting this cold field. For `mcf`, affinity-based reshaping and maximal reshaping result in a performance improvement of about 28%. Both versions of frequency-based reshaping improve `mcf` by about 16%. Maximal reshaping and affinity-based reshaping improve `tsp` by 6.7% and 6%, respectively. For frequency-based reshaping, the improvements for `tsp` range from 5.6% for freq-a to 1.3% for freq-p. The only benchmark where data reshaping results in performance degradation is `perimeter`.[4] All the iterative behavior of `perimeter` results from recursive function calls as it does not contain any loops. Many common optimizations that reduce the cost of array reshaping, such as inlining and loop-invariant expression promotion, are difficult to apply to recursive code. Inlining is important because it enables further local optimizations. It is also difficult to lift common subexpressions from iterative code when the iterations are the product of recursion. A micro-benchmarking study, presented in Section 5.3.3, found that reshaping in `perimeter` significantly increased the number of instructions executed and had little effect on cache efficiency. Therefore, it is not surprising that data reshaping degrades the performance of `perimeter`. Future work will improve the reshaping heuristics to make them more conservative for recursive code.

The performance of array reshaping on POWER4 is similar to that on G5. The superior memory hierarchy on the POWER4 reduces the impact of the poor memory reference in the baseline on the run time of the benchmarks. Therefore, though still impressive, the performance improvement is less significant than that on the G5. The effect of a richer memory hierarchy on the baseline is even more pronounced on the POWER5. As shown in Figure 5.6, freq-a performs better than affinity-based reshaping on `tsp`. This result is the opposite of the findings in Zhong's work [101], where FBP almost always degrades performance for POWER4. Zhong *et al.* split fields that account for 50% of the total accesses into the base object. This threshold should be higher. *Forma* uses 95%. Compared with ABP and MSP, the only advantage of FBP

[4] In `perimeter`, the major data structure of interest is `quad_struct`. There are no infrequently accessed fields in `quad_struct`. Therefore the two versions of frequency-based reshaping choose not to do any reshaping. While this information could be added to the affinity-based reshaper heuristic, it may be desirable to keep the maximal reshaper independent of feedback information.

is that it incurs negligible extra address calculations. A threshold of 50% in FBP generates many extra address calculations and thus eliminates its only advantage. Therefore, it is not surprising that the frequency-based reshaping is much worse than the affinity-based reshaping in [101].

The Itanium-II's architecture is very different from the architecture of the processors in the POWER family. The performance results reflect these differences. The performance improvement for art — 20% for maximal reshaping — is less impressive than that on the POWER family processors. This is because art often iterates on an array with stride 1, which best suits the hardware stream prefetching in the POWER processors. The most significant improvement in Itanium-II is a 45% reduction in run time for the maximal reshaping of mcf. Array reshaping in tsp results in a more significant performance improvement in Itanium-II, from 12.4% to 22.6%, than in the POWER processors. The degradation of perimeter in Itanium-II indicates that the ORC 2.1 optimizations are also limited by the problems caused by recursion.

In summary, for the benchmarks studied in this section, the seemingly naive maximal reshaping performs best or close-to-best in all the studied strategies.

5.3.3 Micro-architecture Performance Study

This section presents measurements obtained with the pfmon performance monitoring tool on the Itanium-II workstation. These measurements further the understanding of the performance impact of data reshaping. This micro-architecture performance study examined the number of retired instructions, the miss rates at different cache levels, the TLB miss rates, and so on. This section presents only the measurements that showed a correlation with array reshaping.

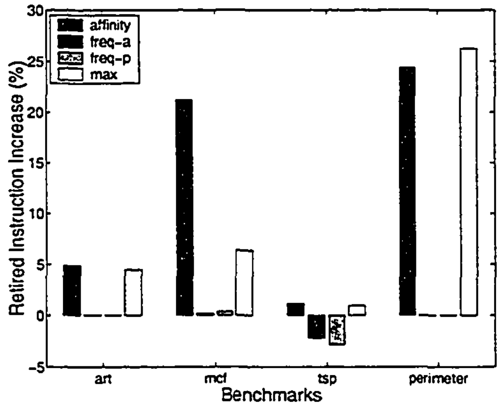
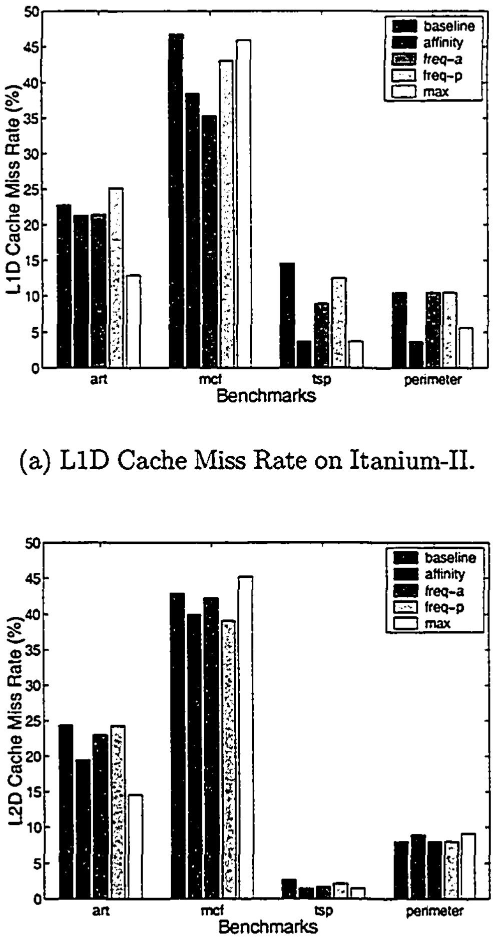


Figure 5.9: Retired instructions on Itanium-II

The number of instructions retired by each version of array reshaping in the Itanium-II workstation, shown as a percentage variation over the baseline in Figure 5.9, provides important insights on the effects of array reshaping at the micro-architectural level. Frequency-based reshaping has negligible instruction increases and results in fewer instructions retired in tsp. These measurements indicate that the profiling input data is indeed representative and this allows the compiler to precisely identify cold fields. Affinity-based reshaping and maximal reshaping may significantly increase the number of retired instructions. The most significant results of this metric are the significant increase in the number of instructions executed for perimeter and the difference in the number of instructions executed for affinity-based and maximal reshaping. A comparison of this data with the execution times in Figure 5.8 reveals a positive correlation with the execution time of the benchmarks.

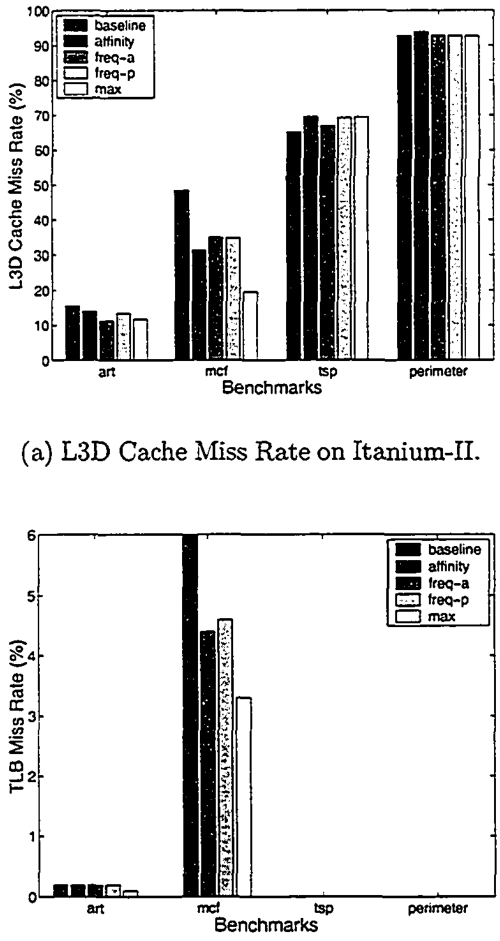


(a) L1D Cache Miss Rate on Itanium-II.

(b) L2D Cache Miss Rate.

Figure 5.10: Data cache (levels 1 and 2) efficiency

Figure 5.10(a) shows the first-level data cache miss rates. Maximal reshaping produces significantly fewer misses in art: the level 1 data cache (L1D) miss rate is reduced by 12.8% compared with baseline. For mcf, frequency-



(a) L3D Cache Miss Rate on Itanium-II.

(b) TLB Miss Rate on Itanium-II.

Figure 5.11: Data cache (level 3) and TLB efficiency

based and affinity-based reshaping reduce the L1D cache miss rate by 21.6% and 15.6%. The effect of maximal reshaping on level 1 data cache in `mcf` is quite small. For `tsp` and `perimeter`, maximal reshaping and affinity-based reshaping achieve similar improvement over baseline.

Figure 5.10(b) shows the second-level data cache miss rates. Different reshaping strategies make little difference on the L2D miss rate for `tsp` and `perimeter`. For `art` and `mcf`, the situation is similar to that in the L1D cache miss rate: maximal reshaping performs best on `art` and frequency-based reshaping is most effective on `mcf`. But the difference of effectiveness on `mcf` is not as significant as in L1D.

The L3D cache miss rates is shown in Figure 5.11(a). There is no significant effect on the L3D miss rate of different reshaping strategies for `art`, `tsp` and `perimeter`. The important data in this graph is the significant reduction, 56.3%, in the L3D miss rate of `mcf` produced by maximal reshaping. With more than 40% access misses on both first- and second level-data cache, at least 16% of `mcf`'s data accesses turn to the third-level data cache. Therefore, the improvement on the L3D cache miss rate avoids a substantial number of memory references (about 6% of the total memory accesses). This improvement is important because the cost of physical memory accesses in this Itanium-II system is about 103.1 nanoseconds, which amounts to about 143 cycles.[5]

TLB efficiency is another important performance factor for benchmarks with large memory footprints. The Itanium-II has two levels of data TLBs (DTLB). To translate a virtual address into a physical address, the processor first looks up the first-level DTLB (L1DTLB). If L1DTLB does not hold the mapping, a four-cycle latency second-level DTLB (L2DTLB) access is required. If L2DTLB also fails, a TLB miss occurs. Itanium-II has a hardware virtual hash page table (VHPT) walker that reduces the overhead of TLB misses [49]. If the translation is not found in the VHPT, the execution traps to the operating system and a software handler is invoked to handle the TLB miss. Software handlers are extremely expensive because they execute hundreds of instructions. Of the four benchmarks studied, `mcf` is the only one that has a relatively high TLB miss rate. However, only about 0.5% `mcf`'s TLB misses result in software handler traps. All the other TLB misses are handled successfully by hardware VHPT walkers. The Itanium-II workstation used in this study has an average TLB miss latency of about 50 cycles.[6] Figure 5.11(b) shows the variations in TLB misses for all benchmarks and reshapers studied. Maximal reshaping reduces the TLB miss rate of `mcf` by 2.3%. This is because maximal reshaping always has the smallest memory footprint. The reduced TLB miss rate combined with the reduced DL3 miss rate (see Figure 5.11(a)) explain the impressive runtime improvement of maximal reshaping for `mcf` in the Itanium-II workstation (see Figure 5.8).

[5] This estimate for the physical memory access latency was obtained with `Lmbench 3.0`.

[6] We used a program at http://www.gelato.unsw.edu.au/IA64wiki/PageFaultTiming to measure TLB miss latency.

5.4 Related Work

Because data cache efficiency is a major performance bottleneck in modern computer systems, extensive research effort has been dedicated to improving data cache utilization. Extant research falls into three categories: data layout optimization, data prefetching, and loop restructuring. This section presents a sample of relevant work in this area.

5.4.1 Data Layout Optimization

The goal of data layout optimizations is to reduce data cache misses by improving data locality or by reducing cache conflicts.

Structure Splitting or Reshaping

Extensive research effort has been dedicated to the study of field placement and data splitting. Using the accumulated frequencies of the member fields, Franz *et al.* split an aggregate data type into a hot structure and a cold one [36]. Zhong *et al.* presents K-distance analysis to group fields in a structure according to their access affinity [101]. Both Franz and Zhong orient their techniques for type-safe programming languages. However, their performance studies use error-prone human-inspected C applications. Applying a type-safe oriented optimization to a type-unsafe language without proper alias analysis is dangerous in production compilers.

Rabbah *et al.* split a structure completely and group the respective fields of various data objects together [71, 73], which is essentially the maximal splitting in this work. In their work, objects may be organized in an array, or may be linked through pointers in the original program. Their research has two shortcomings. First, they use imprecise field-insensitive alias analysis. With this conservative alias information, either important optimization opportunities will be missed or runtime checks must be inserted into the executable, potentially offsetting the data reshaping benefit. Moreover, their analysis does not include safety analysis.

Array Padding and Array Permutation

Inter-array padding adjusts array-based addresses by inserting memory space between arrays while intra-array padding modifies array dimensions by inserting spaces between array elements [74, 50]. The motivation for array padding is to change the array layout so that array elements that are accessed at the same time are not mapped into the same cache address. Array padding is very useful for applications such as dense numerical linear algebra, finite-difference and partial differential equation solvers, and image processing. Array padding and array reshaping work with different data granularities. While array padding treats objects (such as *structs* and *classes*) as atomic, array reshaping works at the field level. Array reshaping is only good for arrays of large aggregate data elements. Moreover, the two techniques have different benefit models: array padding reduces cache conflict misses between frequently referenced data objects while array reshaping tries to avoid bringing useless data into cache.

Strip-mining and array permutation are used to reorganize data in multiprocessor systems to make each individual processor's data share contiguous [52].

5.4.2 Loop Restructuring

Loop restructuring has been used to improve cache efficiency for a long time. Loop fusion might improve cache efficiency if both fused loops have access to the same data elements [55, 63, 81, 96]. Loop fission, also called loop distribution, splits a loop into two or more smaller loops, each of which accesses independent arrays [96]. Loop interchange, also known as loop permutation, reorders the iterations over a multi-dimensional array so that the access pattern is more amenable to data layout [3, 96]. Loop tiling, also referred to as loop blocking, improves cache efficiency by dividing the iteration space of a loop into tiles that have better spatial and temporal locality [45, 75, 95, 96]. Loop tiling can only be applied to perfectly nested loops. Some imperfectly nested loops can be converted to perfectly nested ones so that tiling can be applied. Kodukula *et al.* proposes a "data-centric" approach, called data shackling or data blocking, to localize data accesses [56]. Intuitively, the compiler divides an array into a sequence of smaller blocks, as in loop tiling, and schedules, or *shackles*, the statements that operate on each block close together. At run time, once the data block is fetched into memory hierarchy, the shackled statements are all executed.

All these loop restructuring techniques are compile-time optimizations. These techniques require that access patterns be known to the compiler. However, in some applications the access patterns are hard to predict at compile time. To deal with these situations, researchers have proposed runtime data layout or control flow transformations [31, 86]. However, these approaches have two drawbacks. First, they often ignore practical problems, such as alias analysis for safety, that are indispensable in a production compiler. Second, the optimization targets are often very specific, and reducing the overhead for general runtime transformation is still an open question.

5.4.3 Data Prefetching

Data Prefetching can be seen as an orthogonal optimization to data reshaping. Even though they share the goal of alleviating the memory bottleneck problem by reducing memory latency, data prefetching and data reshaping approach the problem from different angles. Data prefetching is a technique to *tolerate* memory latency by loading data into the cache when the data is expected to be used soon. Data prefetching does not reduce cache misses; rather it reduces the cache miss penalty. Data prefetching tries to handle a cache miss earlier by overlapping the data fetching with other computations so that the data is already in cache when it is needed. Comparatively, data reshaping reduces memory latency by improving data locality and *improving* cache hit rates.

Data prefetching has been studied extensively. A good survey can be found in [89]. Data prefetching can be either hardware based, software based [53, 61, 85] or a joint effort of hardware and compiler support [2, 76].

The drawback of data prefetching is that it may increase substantially the number of memory accesses and the demand for memory bandwidth. This problem becomes dominant in systems where interconnections to a memory subsystem are shared by several processors. From this perspective, data reshaping is superior to data prefetching because it attacks the problem at its cause, by reducing misses, instead of at its observed effect, *i.e.* by tolerating misses.

Badawy *et al.* found that software data prefetching outperforms loop restructuring when there is enough memory bandwidth available [10]. In the same work, they also found that naively integrating software prefetching with loop restructuring does not yield additional performance improvements.

Sometimes the effects of data prefetching and reshaping might overlap. With careful data layout or reshaping, the effect of data prefetching might become smaller because the reshaped data has better locality and is likely to be in cache already when it is referenced, thus eliminating the need for prefetching. If the cache miss pattern can be predicted well by the data prefetching mechanism, data reshaping is not necessary unless memory bandwidth becomes a problem. However, data prefetching and reshaping are not necessarily mutually exclusive. Data reshaping might also facilitate data prefetching. The cooperation between maximal splitting and hardware stream prefetching in PowerPC® processors is a good example.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The research in this thesis is motivated by the observation of heterogeneous reference frequency to instructions and data in many programs. Data and instructions that are not referenced together frequently are nonetheless placed together because programmers tend to write programs that are semantically meaningful. For instructions, codes from the same function do not necessarily have similar runtime frequency. But they may still be placed in the same function because they are closely related to each other. On the other hand, fields in the same data structure are also different in terms of runtime access frequency and pattern. We apply the idea of code and data splitting to reorganize instructions and data. Then we investigate the performance impacts of these splitting techniques. Two important conclusions are drawn in this dissertation.

First, we successfully achieved our goal of function outlining. With appropriate outlining strategies (independent outlining and using alias agent), function outlining reduces the sizes of large functions significantly without performance degradation. However, the main function outlining's client optimization, partial inlining, does not improve performance. The reasons are two-fold. First, the important large functions in the benchmarks studied are so large that they are still too large to be inlined even after they are significantly reduced by function outlining. Second, in applications where inlining was thwarted, usually there are no call sites dominating runtime invocation. Instead, runtime invocation tends to spread in a large range of call sites. The absence of a small set of dominating call sites makes a limited number of partial inlining fail to improve performance. Also, when inlining is conducted too aggressively, its negative impact becomes dominant and we observed large performance degradation.

On the other hand, we observed exciting performance gains through data outlining. Certain benchmarks in SPEC2000 benchmark suite run 2.1 times faster after data outlining. We also investigated different outlining strategies and found that maximal splitting combined with address-arithmetic-based outlining achieves the best or close-to-best performance.

6.2 Future Work

6.2.1 Further Inlining

We identified the two major factors that prevent beneficial inlining in Chapter 3: large function bodies and recursive function calls. We showed that although the problem of a large function body can be ameliorated, it is difficult to be solved by function outlining. It has been shown that only tail recursion can be eliminated by compiler transformation [54]. For other general recursive function calls, it is possible to use recursive function unrolling to reduce runtime function calls. However, because the benchmarks with recursive function calls have large function bodies and no dominating call sites, we believe that the performance impact on SPEC2000 benchmarks will be still very limited.

6.2.2 Extension of Forma Data Outlining Framework.

Forma is a practical array reshaping framework that guarantees safe automatic array reorganization. The experimental evaluation of *Forma* studied the effects of design decisions on two dimensions: the reshaping planner and the reshaping method.

Forma has limitations. Although it catches important cases in standard benchmarks and produces impressive performance improvements, the single instantiation rule for array reshaping is restrictive. Because of this restriction, currently *Forma* only handles dynamically allocated arrays. However, many programs operate on linked data structures that are typically not allocated monolithically into a dynamic array. The next step on the development of *Forma* is to handle these cases. The challenge to handle individually allocated objects is the difficulty of analyzing all the dynamically allocated objects involved in a linked data structure at compile time. Therefore, *Forma* will need to insert a sophisticated memory pool management mechanism into the program and integrate this mechanism into the programs' memory management. Recently, there has been some work in this direction [16, 71, 73]. Adding this feature will increase the number of opportunities for reshaping covered by *Forma*.

A common shortcoming of existing automatic data layout optimization techniques is that they only capture very simple access patterns or use imprecise approximations. To accommodate more complex data access patterns, researchers in the area of cache-conscious algorithms have manually re-engineered applications [44, 68, 69]. Re-crafting an algorithm can produce impressive performance improvements but its high cost makes it prohibitive for a large number of applications. It will be interesting to compare the performance potential of existing automatic reshaping strategies with complex manual transformations. If these manual transformations are general and much more superior, we should investigate more sophisticated program analysis techniques that are necessary to automate this process. These techniques include phase recognition, access pattern (or shape) analysis, and so on. As the impact of memory accesses on performance grows, automatic data reorganization will be justified in spite of its complexity.

6.2.3 Automatic Heuristics Tuning

We manually tuned all the heuristics used in this thesis. Manual tuning not only consume significant time and energy, but also makes it difficult to tell if the heuristics settings exploit the full potential of the optimization techniques studied. This is especially true for function outlining and partial inlining. We used several heristics to control the aggressiveness of function outlining and each heuritic has a very large range of possible values. Consequently, we have a large space to explore to find the best parameters for the heuristics.

Related works have shown that it is feasible to use machine learning techniques to automatically tune heuristics [84]. We believe that similar techniques would better explore the large space of heuristic setting combinations.

Bibliography

[1] http://ipf-orc.sourceforge.net/.

[2] H. Al-Sukhni, I. Bratt, and D. A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 91–100, 2003.

[3] J. R. Allen and K. Kennedy. Automatic loop interchange. In *SIGPLAN Symposium on Compiler Construction*, pages 233–246, 1984.

[4] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Programming Language Design and Implementation (PLDI)*, pages 241–249, 1988.

[5] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Programming Language Design and Implementation (PLDI)*, pages 85–96, 1997.

[6] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Programming Language Design and Implementation (PLDI)*, pages 168–179, 2001.

[7] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive inlining. In *Programming Language Design and Implementation (PLDI)*, pages 134–145, May 1997.

[8] A. Ayers, S. D. Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. In *Programming Language Design and Implementation (PLDI)*, pages 301–312, May 1998.

[9] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 324–341, 1996.

[10] A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In 2001 International Conference on Supercomputing (ICS'01), pages 486–500, 2001.

[11] T. Ball. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems*, 16(5):1399–1410, September 1994.

[12] T. Ball and J. R. Larus. Branch prediction for free. In *Programming Language Design and Implementation (PLDI)*, pages 300–313, 1993.

[13] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[14] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture (Micro29)*, pages 46–57, Dec 1996.

[15] T. Ball and J. R. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, 33:57–65, July 2000.

[16] C. Lattner and V. Adve. Automatic pool allocation for disjoint data structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, pages 13–24, Berlin, Germany, June 2002.

[17] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. In *Journal of Instruction Level Parallelism*, volume 1, pages 22–58, March 1999.

[18] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Principles of Programming Languages (POPL)*, pages 397–408, 1994.

[19] B. Calder, D. Grunwald, D. C. Lindsay, J. Martin, M. Mozer, and B. G. Zorn. Corpus-based static branch prediction. In *Programming Language Design and Implementation (PLDI)*, pages 79–92, 1995.

[20] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 139–149, 1998.

[21] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 60–72, 1996.

[22] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided automatic inline expansion for C programs. *Software - Practice and Experience*, 22(5):349–369, 1992.

[23] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. Ccured in the real world. In *Programming Language Design and Implementation (PLDI)*, pages 232–244, June 2003.

[24] IBM Corp. *C for AIX Compiler Reference*. IBM Corp, International Technical Support Organization, 2002.

[25] J. W. Davidson and A. M. Holler. A model of subprogram inlining. Technical report, Technical Report TR-89-04, Department of Computer Science, University of Virginia, July 1989.

[26] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software - Practice and Experience (SPE)*, 18(8):775–790, 1989.

[27] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering (TSE)*, 18(2):89–102, 1992.

[28] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 77–101, August 1995.

[29] B. L. Deitrich, B. C. Cheng, and W. W. Hwu. Improving static branch prediction in a compiler. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 214–221, 1998.

[30] D. Detlefs and O. Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming (ECOOP)*, pages 258–278, June 1999.

[31] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Programming Language Design and Implementation (PLDI)*, pages 229–241, 1999.

[32] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *The Journal of Instruction-Level Parallelism*, 5, Feb 2003.

[33] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.

[34] P. Feller. Value profiling for instructions and memory locations. Technical report, UC San Diego, U. S. A., April 1998.

[35] D. Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17:614–620, Nov 1974.

[36] M. Franz and T. Kistler. Splitting data objects to increase cache utilization. Technical Report ICS-TR-98-34, Department of Information and Computer Science, University of California, Irvine, Oct 1998.

[37] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[38] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. In *28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 158–168, Dec 1995.

[39] D. J. Hartfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 2:169–192, 1971.

[40] A. Hashemi, D. Kaeli, and B. Calder. Procedure mapping using static call graph estimation. In *Workshop on Interaction between Compiler and Computer Architecture*, Feb 1997.

[41] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization*, pages 253–264, March 2003.

[42] U. Hölzle. *Adaptive optimization for Self: Reconciling high performance with exploratory programming.* PhD thesis, Stanford University, 1994.

[43] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[44] R. C. Holte, T. Mkadmi, R. M. Zimmer, and A. J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1–2):321–361, 1996.

[45] C. Hsu and U. Kremer. A stable and efficient loop tiling algorithm. Technical report, Technical Report DCS-TR407, Department of Computer Science, Rutgers University, 1999.

[46] http://www.redbooks.ibm.com/. *The Power4® Processor Introduction and Tuning Guide.* IBM Corp, International Technical Support Organization, 2001.

[47] W. W. Hwu and P. P. Chang. Inline function expansion for compiling realistic c programs. In *Programming Language Design and Implementation (PLDI)*, pages 246–257, 1989.

[48] Silicon Graphics Inc. *Guide to SGI® Compilers and Compiling Tools.* Silicon Graphics Inc, 2002.

[49] Intel. *Intel® Itanium® Architecture Software Developer's Manual.* Intel Corp, 2002.

[50] K. Ishizaka, M. Obata, and H. Kasahara. Cache optimization for coarse grain task parallel processing using inter-array padding. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 64–76, Oct 2003.

[51] ISO/IEC. *International Standard ISO/IEC 9899, Programming Languages - C. 1st Edition.* 1990.

[52] S. P. Amarasinghe J. M. Anderson and M. S. Lam. Data and computation transformations for multiprocessors. In *Principles of Programming Languages (POPL)*, pages 166–178, July 1995.

[53] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *6th International Symposium on High-Performance Computer Architecture*, pages 206–217, 2000.

[54] O. Kaser, C. R. Ramakrishnan, and S. Pawagi. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems*, 2(1-4):151–164, March–December 1993.

[55] K. Kennedy. Fast greedy weighted fusion. In *14th International Conference on Supercomputing*, pages 131–140, 2000.

[56] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Programming Language Design and Implementation (PLDI)*, pages 346–357, May 1997.

[57] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Principles of Programming Languages (POPL)*, pages 155–169, Boston, MA, Jan 2000.

[58] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *11th International Workshop on Program Comprehension (IWPC)*, pages 33–43, Portland, OR, May 2003.

[59] R. Leupers and P. Marwedel. Function inlining under code size constraints for embedded processors. In *International Conference on Computer-Aided Design (ICCAD)*, pages 253–256, Nov 1999.

[60] C. K. Luk. *Optimizing the cache performance of non-numeric applications.* PhD thesis, University of Toronto, Department of Computer Science, February 2000.

[61] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 222–233, October 1996.

[62] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations.* Wiley, New York, 1990.

[63] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[64] D. Mosberger, L. Peterson, and S. O'Malley. Protocol latency: MIPS and reality. Technical report, TR-95-02, Department of Computer Science, University of Arizona, 1995.

[65] R. Muth and S. Debray. Partial inlining. Technical report, Department of Computer Science, University of Arizona, 1997.

[66] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere. alto : A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, 31:67–101, Jan 2001.

[67] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, pages 128–139, Jan 2002.

[68] R. Niewiadomski, J. N. Amaral, and R. Holte. Crafting data structures: A study of reference locality in refinement-based path finding. In *International Conference on High Performance Computing*, pages 438–448, December 2003.

[69] R. Niewiadomski, J. N. Amaral, and R. C. Holte. A performance study of data layout techniques for improving data locality in refinement-based pathfinding. *The ACM Journal of Experimental Algorithmics*, 9(1–2):17–36, 2004.

[70] P. P. Chang and W. W. Hwu. Trace selection for compiling large c application programs to microcode. In *21st International Workshop on Microprogramming and Microarchitecture*, pages 188–198, Nov 1988.

[71] S. Palem, R. Rabbah, P. Korkmaz V. J. Mooney, and K. Puttaswamy. Design space optimization of embedded memory systems via data remapping. In *2002 Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES'02-SCOPES'02)*, pages 28–37, Berlin, Germany, June 2000.

[72] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.

[73] R. Rabbah and S. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions Embedded Computing System*, 2(2):186–218, 2003.

[74] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Programming Language Design and Implementation (PLDI)*, pages 38–49, 1998.

[75] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th International Conference on Compiler Construction*, pages 168–182, 1999.

[76] A. Roth, A. Moshovos, and G. S. Sohi. Dependence-based prefetching for linked data structures. *ACM SIGPLAN Notices*, 33(11):115–126, 1998.

[77] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, pages 168–179, 2003.

[78] A. Sale. The implementation of case statements in pascal. *Software – Practice and Experience*, 11(9):929–942, September 1981.

[79] R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, Jan 1977.

[80] SGI. Whirl intermediate language specification, 2000.

[81] S. Singhai and K. S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[82] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *6th International Conference on Compiler Construction*, pages 136–150, 1996.

[83] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[84] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta-optimization: Improving compiler heuristics with machine learning. In *Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.

[85] A. Stoutchinin, J. N. Amaral, G. R. Gao, J. Dehnert, S. Jain, and A. Douillet. Speculative prefetching of induction pointers. In *International Conference on Compiler Construction 2001*, pages 289–303, 2001.

[86] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Programming Language Design and Implementation (PLDI)*, pages 91–102, June 2003.

[87] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a Java just-in-time compiler. In *2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, pages 91–104, Aug 2002.

[88] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. In *Programming Language Design and Implementation (PLDI)*, pages 312–323, 2003.

[89] S. P. VanderWiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[90] D. W. Wall. Predicting program behavior using real or estimated profiles. In *Programming Language Design and Implementation (PLDI)*, volume 26, pages 59–70, June 1991.

[91] T. Way. *Procedure restructuring for ambitious optimization.* PhD thesis, University of Delaware, May 2002.

[92] T. Way, B. Breech, and L. L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 24–36, 2000.

[93] T. Way and L. L. Pollock. A region-based partial inlining algorithm for an ilp optimizing compiler. In *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 552–556, 2002.

[94] J. Whaley. Partial method compilation using dynamic profile information. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 166–179, 2001.

[95] M. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

[96] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.

[97] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. Technical Report CS-TR-1994-1248, 1994.

[98] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Programming Language Design and Implementation (PLDI)*, pages 91–103, 1999.

[99] P. Zhao and J. N. Amaral. To inline or not to inline, enhanced inlining decisions. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 405–419, Oct 2003.

[100] P. Zhao and J. N. Amaral. Feedback-directed switch-case statement optimization. Technical Report TR04-26, Department of Computing Sciences, University of Alberta, Edmonton, Canada, 2004.

[101] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Programming Language Design and Implementation (PLDI)*, pages 255–266, June 2004.

Index