Efficient Adaptable Software via Staged Compilation

A Proposal to the NGS/CADSS Program

Project Description

1 Introduction and Motivation

Modern software is becoming increasingly complex. In addition to solving complicated problems, it is asked to be flexible and extensible, i.e., to be easily changed and extended as the problem to be solved evolves. Often the kinds of changes can be anticipated when the original version of the software is developed, but sometimes unanticipated changes must be supported. Software is often asked to run distributed over multiple, possibly heterogeneous machines, and long-running programs may be asked to

One common approach to developing flexible, complex applications is to use a high-level object-oriented language such as Java [Gosling et al. 96]. An object-oriented approach can be applied at the fine grain of individual objects or classes, or at the coarser grain of components. In both cases, the "objects" form convenient units of code organization, helping the programmers manage the complexity of the application. The clear boundaries between objects help to make it easy to replace one object implementation (or group of objects) with an enhanced version. The boundaries also help to make distributed programs easier to write. In some object-oriented languages, including Java, the running program can extend or customize itself with new capabilities (via dynamic class loading in Java), allowing applications to be flexible and adaptable even during their execution. Such run-time adaptability is crucial for long-running applications such as servers and ubiquitous computing infrastructures.

1.1 Dynamic Compilation

In order to keep Java programs portable (i.e., to make them flexible in the face of platform changes), Java source programs are typically compiled statically into a machine-independent intermediate code, Java bytecode [Lindholm & Yellin 97]. Early Java virtual machines "ran" the Java program by interpreting its bytecode, leading to slow programs. To get better performance, later Java virtual machines "ran" the Java program by first *dynamically compiling* its bytecode into native machine code for the particular computing platform on which the Java virtual machine is running, and then executing the machine code. Performance of Java programs is greatly improved by dynamic compilation (also called just-in-time compilation), without sacrificing any of the flexibility, extensibility, and adaptability of Java programs.

However, dynamic compilation as in Java virtual machines comes with serious costs. Most importantly, any time spent dynamically compiling a Java program is time "stolen" from its execution. As a result, dynamic compilers must be fast, even at the cost of optimization quality. Expensive interprocedural analysis, for example, is omitted from current Java dynamic compilers, potentially hurting performance.

In addition to the time cost of dynamic compilation, and its adverse effect on code quality and thus execution speed, there is a space cost for dynamic compilation. The implementation of the compiler itself consumes memory, as do the data structures used by the compiler during and across compilations. The most sophisticated Java dynamic compilers, which produce the highest quality code, typically include several different compilers embodying different cost/quality trade-offs and include extra persistent data structures to guide future compilations [Arnold et al. 00, Whaley 01, Suganuma et al. 01, Pechtchanski & Sarkar 01]; thus the most desirable dynamic compilation infrastructures tend to consume the most space at run time.

1.2 Static Compilation

To overcome these dynamic compilation costs, compilation can be performed ahead of time. Most programming languages are compiled statically, a source file at a time, in the mode of *separate compilation*. Compilers can invest a lot of effort to generate the highest quality code, because none of their time or space costs will be incurred during program execution.

However, the quality of code produced by separate compilation of a single source file can suffer from the absence of information about the rest of the program. In lower-level languages like C, the lack of cross-file knowledge may not significantly degrade compiled code quality for most programs, but for higher-level languages like Java, where the implementation of basic features such as method dispatch and object layout can depend on global properties of the program, separate compilation can be a poor strategy.

To overcome the limitations of separate compilation, the static compiler can be given knowledge about the enclosing program. As one example, interprocedural summaries about the effect of procedures can be computed and then used when compiling callers or callees; this has been used in advanced Fortran and C compilers. In the object-oriented world, various forms of *whole-program* or *link-time compilation* have been investigated [Dean et al. 96, Diwan et al. 96, Fernandez 95]. Such compilers have exploited their complete knowledge of the program being compiled to customize all parts of the compiled application to the particular program configuration. For example, simple knowledge of the program's full class hierarchy can enable much more efficient implementation of method dispatch and subclass testing [Fernandez 95, Dean et al. 95b, Diwan et al. 96], and deeper interprocedural analysis can further reduce the cost of method dispatch [Grove et al. 97, DeFouw et al. 98, Sundaresan et al. 00, Tip & Palsberg 00, Grove & Chambers], optimize object layout [Dolby 97, Dolby & Chien 98, Dolby & Chien 00], optimize memory allocation and eliminate unnecessary synchronization [Aldrich et al. 99, Choi et al. 99, Blanchet 99, Bogda & Hölzle 99, Whaley & Rinard 99, Ruf 00, Gay & Steensgaard 00, Aldrich et al.], and more. Such aggressive optimizations of object-oriented programs are an important complement to their flexibility advantages: programs can be written in a highly modular and malleable fashion, building in the *potential* for change and adaptability into the *source code*, but compiler optimizations can reduce or eliminate the cost of this flexibility for a *particular compiled program* by customizing the compiled code to the particular program configuration being created.

However, static compilation suffers from several limitations. First, because programs are compiled statically into native machine code, the resulting compiled programs are not widely portable, unlike bytecode. Second, it is difficult for statically compiled code to exploit details of the target machine to produce better code, such as cache size or dynamic scheduling strategy, without creating an explosion of versions of compiled code. Finally, and perhaps most importantly, statically compiled programs optimized with whole-program information are no longer as flexible and adaptable as the source code from which they came. To get better performance, flexibility was compiled away, but this typically prevents the compiled program being changed, or extended, or otherwise adapting at run time, such as through Java's dynamic class loading.

1.3 Staged Compilation

We are developing a "third way" that strives to combine the code quality advantages of static compilation with the flexibility advantages of dynamic compilation: staged compilation. In the staged compilation model, parts of a program pass through multiple compiler stages on their journey from source code to optimized machine code. For example, individual source files could first be processed by a separate compilation stage. The main job of this stage would be to resolve as many of the compilation and optimization decisions as possible, given only the information known in the file being compiled; the output of the compiler stage is a partially compiled program representation. The output of several separate-compile stages could then be combined together by a library linking compilation stage. Unlike traditional library linkers, this stage could do additional processing of the linked program parts, such as computing some interprocedural analysis results, and yield further-optimized partially compiled code for the library. A complete-program linking stage could combine partially compiled files and partially compiled libraries into executable programs, performing additional whole-program optimizations in the process. If desired, a run-time compilation stage could perform additional compilation during program execution, for example exploiting details of the run-time hardware platform, or of the program's current usage patterns, or of the program's current class hierarchy.

In general, compilation need not be limited to a single stage, but rather performed in increments over multiple stages. The key advantage of staging is that much of the compilation effort can be shifted to earlier stages, while still enabling information available only in late stages to be exploited for better optimization. Compilation work performed in an early stage is relatively inexpensive, because it can be amortized across the many times that its output is used by later linking and execution stages. Compilation work performed by a late stage is relatively expensive, particularly compilation work performed during program execution, but staging has helped to greatly reduce the amount of work remaining to do in the late stage. In this way, staging strives to combine low late-stage compilation effort with high compiled code quality.

Staging contributes to greater adaptability and flexibility of high-performance software. Programs can be well optimized to the current run-time usage context, more than would be possible with purely static compilation, for instance exploiting current execution profile information, or program configuration information (such as the set of classes that have been dynamically loaded), or hardware configuration. But if these conditions change, the run-time compilation stage can be rerun, to produce new compiled code optimized for the latest run-time conditions. In this fashion, the executing compiled code can be nearly continuously adapted and optimized to the current execution conditions. Because this run-time adaptability is supported by the compilation and execution infrastructure, programmers do not have to work as hard to build run-time adaptability into their software manually. Moreover, programmers can use modern software development approaches, such as modular, object-, or component-oriented programming, with less fear of receiving poor-quality compiled code.

1.4 Proposed Work

We propose to develop staged compilation into a practical approach to compilation, and to experiment with its utility on several large, complex software systems. Many theoretical and practical challenges remain to be overcome before staged compilation is an effective and convenient approach for many applications. The main challenges we will address are the following:

- The construction of staged compilers is difficult and error-prone. We propose to develop techniques to mechanically generate staged compilers from simpler unstaged ones. We have already received encouraging results from a proof-of-concept system we developed. We have identified many additional techniques that we hope will further improve the precision, cost, and utility of mechanical staging.
- Only a few, simple optimizations have been staged. We propose to investigate a wide range of compiler optimizations, and in particular those that target higher-level object-oriented languages and those that apply in distributed and parallel contexts, across a range of input programming languages.

Early staged compilers have tackled only small programs. We propose to investigate how well staging can be applied to several large (up to hundreds of thousands of lines), complex, long-running, object-oriented applications. These applications include parallel execution issues, compact object layout optimizations, distributed systems, and dynamic program evolution. These applications need to combine high performance, modular software organization, and adaptability (either statically or a run-time). We feel they are representative of a broad class of programs that are important to execute well, and current compilation techniques yield unsatisfactory results.

1.5 Outline

The next section of this proposal describes our previous work on advanced compilation systems. Section 3 then describes our proposed work in detail, and section 4 discusses our target applications. Section 5 discusses the overall expected impact of this project and how it relates to the Next Generation Software program. Section 6 identifies project milestones, and section 7 describes our management plan. Section 8 reports on results from the last five years of our previous NSF-funded research.

2 Our Previous Work

Our previous work has provided us with the necessary understanding of the technical issues and the implementation experience to tackle this proposed research. One of us (Craig Chambers) developed one of the first dynamic optimizing compilers, the Self compiler, which pioneered many techniques in dynamic compilation and optimizations for object-oriented languages; this work is described in section 2.1. Chambers also developed one of the first whole-program optimizing compilers aimed at object-oriented languages, the Vortex compiler. This work is described in section 2.2. These two projects have given us experience in the two main compilation models whose strengths we wish to combine via staged compilation, and they also have given us knowledge of how to optimize object-oriented programs well. Together we developed one of the first selective run-time optimizers, the DyC (pronounced “dicey”) optimizing compiler for C. DyC selectively applies optimizations at run-time, most of which are examples of staged optimizations: the static compiler stage performs optimization preplanning ahead of run time, producing for each portion of the program being dynamically compiled a specialized run-time compiler stage. When invoked at run time, the specialized run-time compiler quickly completes the compilation process, producing high-quality target code at low run-time cost. Section 2.3 describes this work in more detail. DyC’s work is guided by user annotations. Recently we have developed the Calpa system for automating the selection of where to dynamically compile, hiding staging completely from the user. Section 2.4 describes this work.

2.1 The Self Optimizing Compiler

The Self compiler was the first highly optimizing run-time compiler for an object-oriented language. It extended earlier unoptimizing run-time compilers for Smalltalk [Deutsch & Schiffman 84] with a suite of aggressive analyses and optimizations:

- Class analysis tracks the set of possible classes of variables, in order to optimize dynamically dispatched message sends and similar run-time class-based operations [Chambers & Ungar 90].
- Customization produces multiple copies of source methods, each specialized for a particular receiver class and thereby facilitating class analysis of the receiver formal argument [Chambers & Ungar 89, Chambers et al. 89]. Run-time compilation can be exploited to compile customized versions only on-demand, as particular methods are invoked on particular receiver classes.
- Run-time compilation can also be exploited to compile only on demand any control subpaths within methods that are expected to be rarely needed in practice, such as error handlers [Chambers & Ungar 91]. This strategy enables features, such as checked array bounds and automatic numeric overflow handling, found in higher-level languages, to have little cost in practice.
- Inlining is performed automatically for calls of small, non-recursive methods. To make better automatic inlining decisions, estimates of the cost and benefit of post-inlining optimizations are considered [Dean & Chambers 94].

- The compiler consults the run-time message dispatching structures to learn which receiver classes occur most commonly at different call sites. This on-line profile feedback (hints of likely classes to be used during class analysis) is exploited when dynamically recompiling heavily used methods, consequently improving performance [Hölzle et al. 91, Hölzle & Ungar 94a]. In addition, by restricting optimization effort to heavily used methods, overall compilation time can be greatly reduced [Hölzle & Ungar 94b].
- To make these advanced compilation techniques compatible with an interactive development environment, Self includes mechanisms that support full source-level debugging of optimized code [Chambers et al. 89, Hölzle et al. 92] and selective recompilation after programming changes [Chambers et al. 89].

For the purely object-oriented language Self, these optimizations provided speed-ups of an order of magnitude for medium-sized benchmark programs. Many of the optimizations make critical use of run-time compilation; traditional static, separate compilation would prevent the demand-driven compilation techniques, profile-feedback-based optimizations, and class hierarchy knowledge that enables compile-time resolution of method lookup even in the face of potential changes to the program at run-time. The basic techniques developed in the Self compiler, such as class analysis, form the foundation for most later work on optimizing compilers for object-oriented languages. The run-time-compilation-based techniques pioneered in Self have been the basis for much of the current work on dynamic compilers for Java [Arnold et al. 00, Whaley 01, Suganuma et al. 01, Pechtchanski & Sarkar 01].

2.2 The Vortex Optimizing Compiler for Object-Oriented Languages

The Vortex compiler adopts a link-time, whole-program compilation model. This approach avoids the space overheads and harsh compilation time constraints of Self's run-time compilation model, while still preserving complete information about the program being compiled. Vortex's form of class analysis exploits complete knowledge of the program's class hierarchy [Dean et al. 95b]. Vortex includes a variety of interprocedural class analysis algorithms that represent different trade-offs between precision and cost of analysis [Grove et al. 97, DeFouw et al. 98, Grove & Chambers]. Vortex preserves most of the benefits of Self's dynamic profile feedback through an off-line profiling technique, where dynamic profile information gleaned from a specially instrumented executable is fed back to the Vortex compiler for use in future compilations [Grove et al. 95]. Similarly, Self's demand-driven method customization strategy is generalized in Vortex by using a combination of static class hierarchy information and dynamic profile information to guide selective procedure specialization [Dean et al. 94, Dean et al. 95a]. Whole-program knowledge is exploited in more specialized ways in Vortex as well, for example, to produce an efficient implementation of message dispatching stub functions [Chambers & Chen 99] and to eliminate unnecessary synchronization in Java programs [Aldrich et al. 99, Aldrich et al.]. To make link-time whole-program optimization practical to use in a normal program development environment, Vortex includes automatic dependency tracking mechanisms that enable selective recompilation of only affected modules after program changes [Chambers et al. 95]. Vortex also includes internal frameworks to ease the task of defining and composing intraprocedural and interprocedural analyses [Chambers et al. 96].

The Vortex compiler supports several different language front-ends, ranging from lower-level languages like C and C++ [Stroustrup 91] through hybrid object-oriented languages like Java to high-level, purely object-oriented languages like Smalltalk [Goldberg & Robson 83] and Cecil [Chambers 93a, Chambers 96]. All front-ends translate into a common intermediate language, which is optimized uniformly across languages. This range of front-ends enables experiments of the effectiveness of optimizations to be performed in a wider variety of benchmarks and programming styles than in other compilers [Dean et al. 96].

To assess the effectiveness of these optimizations, we collected a suite of C++, Java, and Cecil benchmarks, each 10,000-60,000 lines in size. Overall, the techniques studied in Vortex yield speed-ups on our benchmarks of 30-40% for C++, 50-80% for Java (over straightforward translation to C and subsequent optimized compilation of the C code, which is a factor of four better than interpreted Java and a factor of two better than typical Java just-in-time compilers), and a factor of more than seven for Cecil.

The Vortex compiler system has been in daily use internally for its own development since early 1994, helping to ensure that the techniques used in Vortex are practical in a normal program development environment. The entire Vortex system, including front-ends for C++, Java, and Cecil, and back-ends for portable C code and SPARC assembly code, was originally released to the outside research community in April 1997, and a second major public release was made in February 1999.

2.3 The DyC Run-Time Compilation System for C

DyC compiles and optimizes programs dynamically, during their execution [Auslander et al. 96, Grant et al. 97, Grant et al. 99, Grant et al. 00a, Grant et al. 00b]. To trigger run-time compilation, programmers annotate their source code to identify *static variables*, i.e., variables that have a single value (or relatively few values) during program execution, and on which many downstream calculations depend. DyC then automatically determines which parts of the program downstream of the annotations can be optimized based on the static variables' values (we call these *dynamic regions*), and arranges for each dynamic region to be compiled at run-time, once the values of the static variables are known. To minimize run-time compilation overhead, DyC does much of the analysis and planning for run-time compilation and optimization during static compile time.

DyC extends a traditional (static, separate) optimizing compiler with two major components, as illustrated in Figure 1.

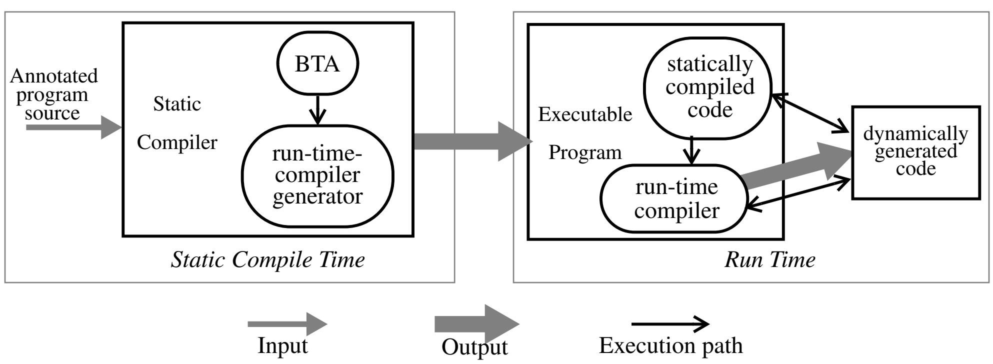

Figure 1: DyC's Static and Dynamic Components

- A *binding-time analysis* (BTA) identifies those variables (called *derived static variables)* whose values are computed solely from other annotated or derived static variables and determines the extent of each dynamic region. It then divides operations within a dynamic region into those that depend solely on static variables and therefore can be executed only once (the *static computations*) and those that depend at least in part on run-time data and must be reexecuted each time the flow of execution reaches them (the *dynamic computations*).
- A *run-time-compiler generator* produces a fast run-time compiler that will generate code at run time for each dynamic region, once the values of the static variables have been computed.

In more detail, DyC performs the following steps when compiling each procedure at static compile time:

- First, DyC applies many traditional intraprocedural optimizations, stopping just prior to register allocation and scheduling.

- Then, for procedures that contain annotations, the binding-time analysis identifies derived static variables and the boundaries of dynamic regions. This analysis also determines which conditional branches and switches can be completely resolved and which loops can be completely unrolled at run-time, once the values of the static variables are known.
- Each dynamic region is divided into two control flow subgraphs, one containing the static computations (called *set-up code*) and one containing the dynamic computations (called *template code*). Where a dynamic instruction in the template code refers to a static operand, a place-holder operand (called a *hole*) is used. The hole will be filled in at run time, once its value is known.
- Register allocation and scheduling are then applied to the two subgraphs separately.
- Finally, a custom run-time compiler for each dynamic region is built by inserting *emit* code sequences for each instruction in the template code into its corresponding position in the set-up code (the template subgraph is then thrown away). This run-time compiler is fast in large part because it neither consults an intermediate representation nor performs any general analysis when run. Instead, these functions are in effect "hard-wired" into the custom compiler for that region, represented by the set-up code and its embedded emit code.

At run time, when a dynamic region is first entered, its custom run-time compiler is called. The run-time compiler first checks its local dynamic-code cache for a previously compiled version for the values of the static variables on entry, and reuses it if found. Otherwise, it continues executing, evaluating the static computations and emitting machine code (filling holes from the just-computed static values and performing run-time strength reduction in the process). The generated machine code is saved in the dynamic region's cache, and then executed.

The run-time compiler can apply multiple optimizations: run-time constant propagation and folding, run-time strength reduction, run-time conditional branch folding and dead-code elimination, run-time complete unrolling of loops (including those that unroll into arbitrary directed graphs of loop bodies, which we call *multi-way loop unrolling*), run-time copy and zero propagation (important for architectures like the Alpha where floating point moves and clears (derived, for instance, from strength-reducing a multiplication by a static variable whose run-time value is 1.0 or 0.0) cost as much as any other floating point operation), and run-time dead-assignment elimination. Each of these run-time optimizations is staged, with some advance preparation work performed statically, to greatly reduce its run-time cost.

Though still a prototype, our experiments have shown that run-time compilation with DyC can be very profitable: we obtain speed-ups as high as 4.7 on real applications. Each of DyC's optimizations was important for at least one of our benchmarks in achieving good speed-ups.

The work on DyC gives us experience in using multi-stage optimizations to increase optimization opportunities at low cost. Our proposed work will build on this experience, extending it to develop multi-stage versions of other optimizations (such as object-oriented class analysis), to include compilation stages between separate compilation time and run time, and to base optimization decisions on the detailed characteristics of the underlying hardware.

2.4 The Calpa Automatic Run-Time Compilation System for C

DyC requires that programmers add annotations to their C programs to indicate where and how to apply dynamic compilation. We have recently developed a "front-end" to DyC, named Calpa, that automates this selection process [Mock et al. 99, Mock et al. 00]. Calpa first uses a static pointer analysis to compute an approximation of what each pointer source variable might point to. It then inserts instrumentation into the program and runs it on sample input (provided by the user) to gather *value profiles* [Calder et al. 97, Sodani & Sohi 98]. Value profiles indicate which variables and data structure contents are frequently referenced and have only a single or just a few run-time values. Such variables are excellent candidates for selective dynamic compilation by DyC. Given the value profiles and the static pointer analysis, Calpa performs a cost/benefit analysis on the source program to estimate, for each possible combination of annotations of variables, whether the expected benefits of selective dynamic compilation outweigh its expected run-time compilation costs. Calpa chooses the best annotation combination, adds the annotations to the C program, and passes the annotated C program to DyC for compilation. For the DyC benchmarks, Calpa has been able to generate automatically the annotations originally determined by hand, plus a few additional annotations in some benchmarks. Current work is seeking to apply Calpa to other benchmarks not considered in the DyC because of the laborious nature of determining the right annotations by hand.

Calpa gives us experience in automating parts of staged compilation, and in providing profile feedback from later stages to earlier stages. More aggressive application of these ideas, to a wider range of stages and a wider range of staged optimizations, is part of the proposed work.

3 Proposed Work

We propose to develop staged compilation into a practical approach to compilation, and to experiment with its utility on several large, complex software systems. Many theoretical and practical challenges remain to be overcome before staged compilation is an effective and convenient approach for many applications. The main challenges we will address are the following:

- Our experience with DyC has convinced us that the construction of staged compilers is difficult and error-prone. We propose to develop techniques to mechanically generate staged compilers from simpler unstaged ones. We have already received encouraging results from a proof-of-concept system we developed, which is described in section 3.1. We have identified many additional techniques that we hope will further improve the precision, cost, and utility of mechanical staging, described in section 3.2.
- Only a few, simple optimizations have been staged. We propose to investigate a wide range of compiler optimizations, and in particular those that target higher-level object-oriented languages and those that apply in distributed and parallel contexts, across a range of input programming languages. These efforts are described in section 3.3.
- Relatively little work has studied how staging applies to different uniprocessor and multiprocessor platforms. We propose to investigate these interactions, as described in section 3.4.

Section 4 describes our target applications with which we will evaluate and guide our work.

3.1 SCF: An Automatic Stager

A significant obstacle to the wider use of staged compilation is the amount of effort it takes to develop staged compilers. Traditional single-stage compilers are difficult enough, but the difficulty of writing an early compiler stage is another level entirely. This is because the early stage is given only partial information about the program to be compiled and is required to perform as much compilation as possible given this partial information. The late compiler stage must then take as input partially compiled programs and then complete the optimizations as quickly and effectively as possible. To make this work in the DyC compiler, the static compiler stage had to reason about the possible effects of optimizations that depend on information that will only become known at the run-time stage. In effect, the early stage had to perform a kind of case analysis of all the ways that optimizations might proceed at run-time, when given complete information, and then precompute for each case as much of the final result as possible. This "meta" reasoning about the behavior of the optimizations themselves rather than just the program to be compiled was very difficult to implement correctly and effectively by hand.

To address this difficulty, we have developed an initial proof-of-concept system, called SCF for "staged compilation framework," that can automatically and mechanically construct a staged compiler from the implementation of a single-stage optimizer. The compiler writer, i.e., the user of the SCF system, first

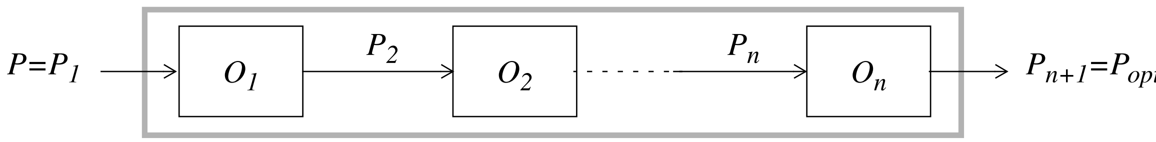

Figure 2: An Optimization Pipeline

writes a series of program optimizations $O_1..O_n$ that takes in a program P to produce an optimized program P_{opt}, as illustrated in Figure 2. Each optimization in SCF is written in a side-effect-free subset of ML. Each optimization O_i has the form of a program transformer that takes in an arbitrary ML data structure P_i (typically a tree representation of the procedure to be optimized and possibly additional context information about the properties of the formal parameters) and produces a transformed ML data structure P_{i+1}. The output of one optimization is then fed in as input to the next optimization, in a classical pipeline structure. Writing these optimizations should be the same work as writing an optimization pipeline in a regular unstaged compiler; the optimization writer need not be aware that the optimizations are to be staged.

To get the highest quality of compiled code, the optimization pipeline should be run at the latest possible moment, i.e., at run time, when it can observe the whole program being run and its execution context. But this would normally be prohibitively expensive to delay all compilation and program analysis until run time, unless optimization quality is greatly sacrificed. But before run time, much is known about what the program and run-time context will be. For example, at program link time, most if not all of the program text will be known, and perhaps some information about the target platform (such as its instruction set architecture) will be known. In this case, it is possible to precompute much of the result of the run-time compilation ahead of time, at program link time. The SCF system does this by *partially evaluating* [Jones et al. 93] each of the optimizations given a *partial description* of its input. The component of SCF that does this work is called the *stager*.

The partial description takes the form of a set of possible ML data structures that might be given to the later stage. In our application of staging compiler optimizations, the "set of possible data structures" might describe families of program trees that might be invoked at run time, and/or sets of possible configuration parameters for the run-time environment or target platform (e.g., the cache size or number of processors), and/or sets of possible classes that might be dynamically loaded, and/or sets of possible values to be provided by user input at run-time. The SCF system does not depend on the meaning of the inputs and outputs of the optimizations, giving it great flexibility to be used in a variety of staging applications.

One result of applying SCF to an optimization is a partial description of the optimization's result data structure; this description is then used to apply SCF to the next optimization in the pipeline. The other result of applying SCF to an optimization is a *specialized optimization*. The specialized optimization is what is left of the original optimization after all the parts of its work that can be precomputed have been; it is the work that remains to be done to complete the optimization when it is finally given the complete program and context information. When staging a whole pipeline, a pipeline of specialized optimizations is produced. The specialized pipeline can then be run just like the original pipeline, with exactly the same results, as long as its input is a member of the set of inputs described by the partial information.

Figure 3 illustrates the effect of staging an initial pipeline of optimizations ${}_1O_1..{}_1O_n$ with respect to partial information I to produce a specialized optimization pipeline ${}_2O_1..{}_2O_n$. (We will use prefix subscripts to

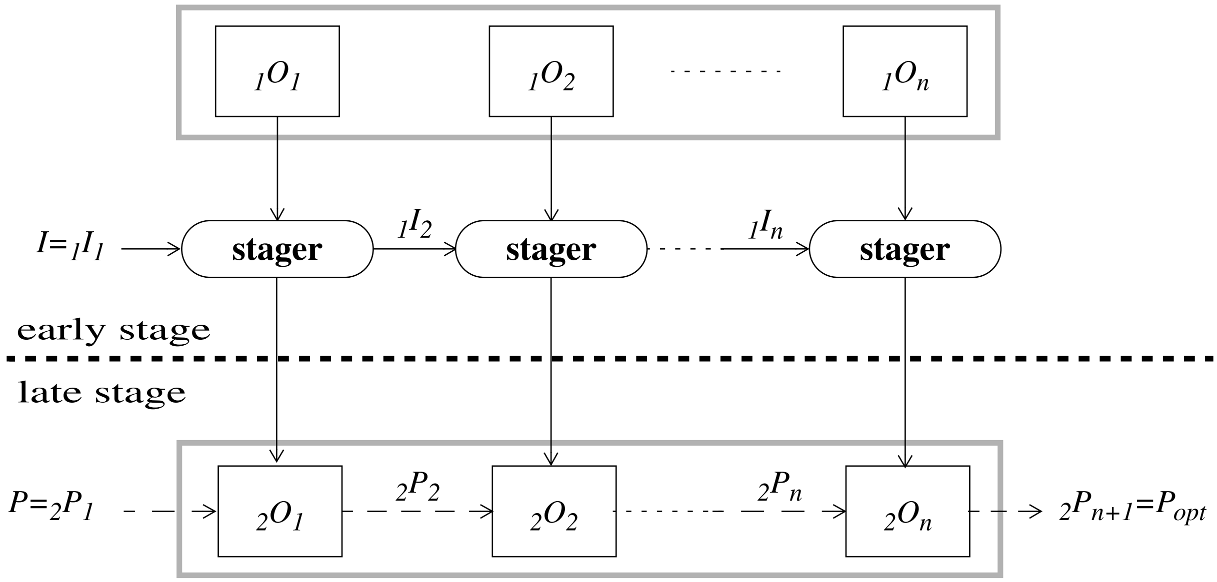

Figure 3: Staging of a Pipeline

refer to the stage of the optimization, the partial information, or of the program being optimized, i.e., the row in the diagrams. Suffix subscripts will refer to the position in the phase-ordered pipeline, i.e., the column in the diagrams.) The partial information I describes the set of possible input programs on which the specialized optimization pipeline $_2O_1..{_2O_n}$ may be invoked. Staging corresponds to execution of the early stage. The stager does not *run* its input optimization, but rather takes the *code* of the optimization as input to produce the *code* of the specialized optimizer. In the late stage, the specialized optimization pipeline is *run* on an input program P to produce an optimized program P_{opt}; P must be a member of the set described by I. The late stage may be run many times on different inputs P, thus amortizing the work of staging. Moreover, by precomputing much of the work of each optimization, specializing the optimization pipeline to just those programs described by I, the stager enables the specialized optimization pipeline to run much faster than would the original pipeline.

The specialized optimization pipeline can itself be staged to produce a yet-more-specialized pipeline, given more refined but still incomplete information about the set of possible programs. This staging process can be repeated as many times as desired. Each stage corresponds to a time during program development or execution when additional program or context information becomes available. For example, the first stage could correspond to separate compilation time, later stages to library linking time, a late stage to complete program linking time, a very late stage to program load time (the start of program execution), and an even later stage to points during program execution where key values manipulated by the program are computed. Figure 4 illustrates the general form of SCF-based staged compilation applied to a pipeline of n optimizations processed over m compilation stages. At each stage but the last, the *code* of an optimization is consumed by the stager to produce the *code* of the specialized version of optimization to be used in the next stage; only in the last stage is the final specialized optimization pipeline actually invoked on an input program to produce a compiled output program.

Early experimental results using SCF show that automatic staging can give order-of-magnitude speedups to optimization pipelines (the residual optimization pipeline can be an order of magnitude faster than the original pipeline) at a reasonable early-stage compilation time cost. More details about the techniques

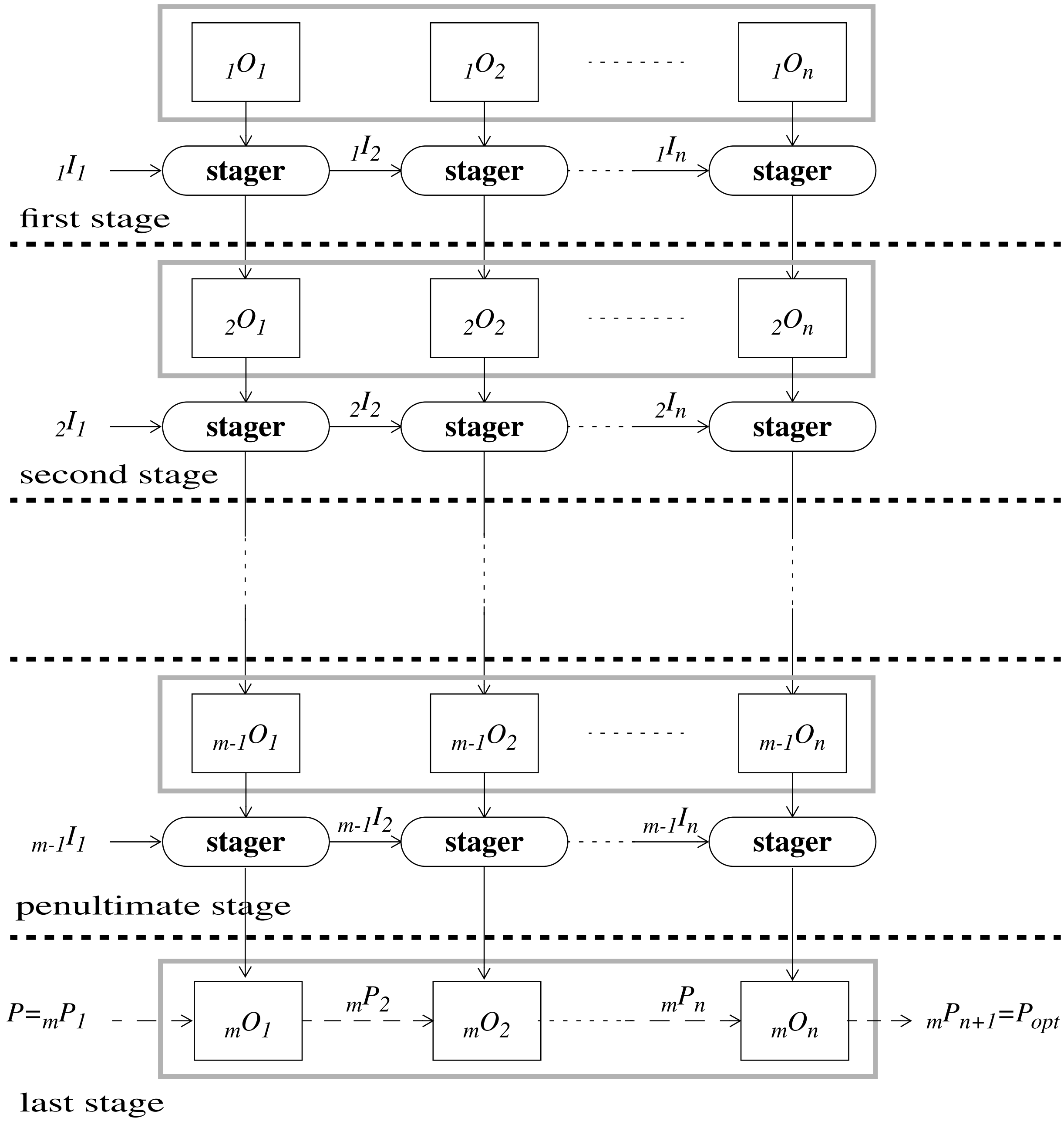

Figure 4: General Scenario of Staged Compilation

used in the SCF stager are available elsewhere [Philipose et al. 02].

3.2 Extensions to the Staging Framework

SCF is a proof-of-concept, demonstrating that automatic construction of staged optimizers is feasible. This success inspires us to tackle an array of issues necessary to “scale up” staging to be usable in practice over a wide range of optimizations, stage times, and user constraints.

Several directions strive to make the specialized optimization pipelines execute more quickly (thereby lowering the cost of late-stage optimization):

- The stager includes a sophisticated approach to context-sensitive analysis and specialization, but further improvements in analysis precision are desired. In particular, sometimes the stager's analysis merges control flow paths in the optimizer being analyzed, causing the stager's abstract model of the contents of variables along the two paths to be intermingled. We plan to investigate a range of techniques for keeping paths separate when it is important (but not too often), ranging from heuristics for eagerly splitting paths apart when they contain significantly different information, to techniques for lazily splitting paths apart when more precise analysis information would be helpful [Chambers & Ungar 90], to techniques to better track cross-variable correlations of the abstract models of variable contents.
- Because of its roots in partial evaluation technology, the current stager produces specialized optimizations that take in complete programs. This is not necessary, since many aspects of the input program are hard-wired into the specialized optimizations. We will develop techniques for passing in only the parts of the programs that remain unknown to the specialized optimization, and for having the specialized optimizations manipulate this more compact program representation throughout the specialized pipeline. By analogy, DyC inputs only the run-time values of the static variables, not the whole program with those values plugged in as constants. We wish to imbue the automatic stager with this ability (albeit in a much more general context). This work has general applicability to partial evaluation of partially static data structures; our techniques are a way of compactly representing only the dynamic parts of a partially static data structure.
- The specialized optimizers currently produce the complete optimized output program at run-time. However, just as much of the input to specialized optimizers is known ahead of time, much of their output is known. We will develop techniques for "pre-executing" the specialized optimization pipeline to produce precomputed templates of parts of the output optimized program; the effect of running the specialized pipeline would be to select from the precomputed templates and fill in any holes, much as DyC does (although in a more general form, and automatically).
- The current stager generates specialized optimization pipelines that produce exactly the same optimized program as would the original optimization pipeline, if given the same late-stage input information; shifting some of the compilation effort to an early stage does not hurt the quality of optimization of the late stage. While desirable, meeting this standard of "complete meaning preservation" requires that any optimization that might conceivably be performed at run time must be included as part of the specialized compiler. This can make the specialized optimizers be very large and somewhat slower, burdened with the extra work of checking for many theoretically possible but unimportant optimization possibilities. A more practical specialized optimizer would sacrifice minor optimization opportunities if doing so allowed the specialized optimizer to be much smaller and faster. For example, if the specialized optimizer can perform its work in a single pass, without any iterative dataflow analysis, then it can avoid maintaining a full intermediate representation of the program, allowing it to be much faster and smaller than a traditional optimizer [Consel & Noël 96, Lee & Leone 96, Poletto et al. 97, Grant et al. 99]. We will investigate techniques for (optionally) ensuring that specialized optimizers can execute in a single pass, and more generally for performing a cost/benefit analysis on whether some optimization step should be delayed until the later stage (put into the specialized optimizer) or simply resolved conservatively in the early stage (removing it from the specialized optimizer).

Another direction investigates an alternative approach to producing specialized optimization pipelines:

- The current model of staging produces a optimization pipeline specialized to the particular set of programs that the pipeline might be invoked on. This produces the *fastest* specialized pipelines, but also the *largest*. Once specialized optimizations become too large, they can slow down due to poor instruction cache behavior. An alternative approach is to specialize the program representation, not the optimization pipeline. For example, an early separate-compilation stage could construct a file's dataflow graph and perform many intraprocedural and intra-file optimizations using it. Instead of producing a specialized optimization pipeline to use as the next stage, the early stage could produce a specialized representation of the file, for example including the portion of the dataflow graph that connects to procedure formals and results and that accesses external variables. The next compilation stage, e.g. the library linker, can combine this specialized, partially compiled representation with the specialized representations of other files to quickly optimize the linked files in their combined context. Specialized representations can be much more compact than whole specialized optimization pipelines, in part because a general-purpose optimization pipeline can be used to compile all specialized representations, but the later stages are probably slower than they would be if a specialized pipeline can been constructed. Developers of staged compilers can mix the two specialization techniques to get the right balance of speed and compactness; for example, to save space the static stages might all use specialized program representations, but for highest performance the run-time stage might be a specialized optimization pipeline. We will investigate techniques for automatically transforming a regular unstaged optimization pipeline into a pipeline that can input a specialized, partially compiled program representation and output a further specialized, more optimized representation.

Other directions seek to make staging more controllable and understandable:

- A feature of SCF is that the optimizations being staged do not need to be written with staging in mind. However, this feature can also be a limitation, because the optimizations have little control over how they are staged. We will study techniques for giving optimizations limited ability to query how they are being staged, for instance in the form of reflective questions about whether the values of data structures being manipulated by the optimization are known during staging. For example, a constant folding optimization might go to great lengths to try to find an efficient way to multiply an integer variable by a constant, using combinations of shifts and adds, as long as the constant is actually known at staging time. If, however, the constant is only known to be "some constant, to become known in a later stage," then the optimization writer may wish to avoid the overhead of delaying all that work until the late stage, and so may be willing to skip optimizing the multiply instruction. (This is an example of sacrificing some minor optimization opportunities in order to make the specialized optimizer faster, but under control of the optimization itself.) More advanced optimizations like object-oriented class analysis may have even greater need to be written with some limited awareness of how they are likely to be staged. For example, references in the optimization to information known only with global knowledge may need to be handled more carefully, since this information will only be known fully in a late stage.
- We also will develop tools for enabling the compiler developer that uses our staging infrastructure to examine the effects of staging, such as which operations in the original optimization were able to be precomputed in the early stage, and how many specialized versions of the rest of the operations were generated in the specialized optimizer.

Other directions strive to integrate staging into an interactive, feedback-driven compilation infrastructure:

- Late-stage optimizations are run (or further staged) given information known about the program or execution context at that stage. This information may change over time, either as the program execution patterns change, or the program itself is changed, e.g. through dynamic class loading or static program development and maintenance. When the information changes, the affected stages must be re-run on the new information. We will develop dependency frameworks that record enough information to trigger reexecution of a stage whenever the information it depended on changed, based on the dependency frameworks used for a similar purpose in the Self, Vortex, and Jalapeño systems [Chambers et al. 89, Chambers et al. 95, Pechtchanski & Sarkar 01] and the invalidation mechanism in DyC [Mock et al. 00]. For example, when a class is dynamically loaded, the dependency framework would trigger the reexecution of any run-time stages that depended on the old version of the class hierarchy, or more specifically, on any code that invoked methods that have now been overridden by the newly loaded class. As an alternative mechanism, we also will experiment with inserting run-time guards at the start of code generated by late-stage optimizers to verify that the values given to the late-stage optimizers are still valid, as is done with DyC.

- Run-time profile information can be exploited through SCF by delaying until run time the parts of optimizations that depend on the profile data. However, it might be useful to provide a description of the likely possibilities to early stages, so more of the work of optimizing using the profile data can be shifted to earlier stages. We will investigate techniques for profiding profile feedback to earlier stages, e.g., in the form of additional partial information about the program, so that the optimizations that read run-time profile data can be staged using the partial information. One twist that distinguishes this from regular staging is that the profile data is usually treated as merely a *hint* as to likely program execution behavior, while the partial information exploited by staging above was assumed to be a *guarantee* about the characteristics of the eventual program to be compiled by the last stage. However, in the face of dependency frameworks and guard tests, the line between hints that might be wrong and guarantees that may be invalidated or guarded for becomes blurry. We will seek a common set of mechanisms that handles both guarantees and hints well.
- Similar feedback can occur between static stages as well. For example, the complete-program linking stage has knowledge about the complete class hierarchy of the program being linked (at least in the absence of Java-style dyanmic class loading). This information would be useful in earlier stages, enabling them to precompute many of the possible results of method lookups. The later stage could then just select from among the precomputed alternatives, or, if the predicted class hierarchy was wrong, fall back to running the class-hierarchy-based optimization during the later stage. We will develop mechanisms for feedback between static stages, ideally using the same kinds of mechanisms as dynamic feedback.
- Profile information is also helpful to identify where it will be most profitable to defer optimization until run-time. Calpa uses value profiles to identify where DyC's optimizations could be profitably done at run-time. We will study more general techniques for using profile information to guide where it is worth delaying an optimization operation to the later stage, complementing the optimization-based controls of staging described above.

Overall, these techniques seek to develop the initial proof-of-concept ideas in SCF and Calpa into a general, effective, practical compilation infrastructure.

3.3 Staging More Optimizations

SCF and other selective dynamic compilers have only investigated staging of a few optimizations, such as constant propagation, copy propagation, and dead-assignment elimination. A key goal of mechanizing staging is to be able to apply staging to a wide range of optimizations.

- SCF now stages target-machine-independent optimizations. We wish to experiment with staging target-machine-dependent optimizations, including instruction scheduling, register allocation, and machine code generation. Staging these optimizations will allow a run-time specialized optimization pipeline to generate machine code directly into memory at run-time, an important ability of all the previous hand-written selective dynamic compilers, including DyC.
- For our target domain of advanced object-oriented applications, we are particularly interested in studying how best to stage class analysis [Johnson et al. 88, Chambers & Ungar 89, Chambers & Ungar 90], which is the fundamental analysis of object-oriented programs. Knowledge of the possible classes of variables allows the results of method lookup to be precomputed, which in turn enables inlining of known methods and additional interprocedural optimizations that depend on knowledge of the program's call graph. Information about the possible classes of variables becomes available to the compiler stage-by-stage.
    - For example, in the separate-compilation stage, simple class information about the results of object instantiations is known and can be exploited, but little else can.

- In each of the library linking stages, objects flowing across classes but within libraries can often be determined to be instances of a small, bounded set of classes, enabling optimizations of those parts of the program to be performed. The separate-compilation stage can have precomputed each class's part of the interprocedural dataflow graph, helping to make the library linking stage run fast.
- In the complete-program linking stage, complete knowledge of the program's class hierarchy becomes available (in the absence of dynamic class loading), and much of the possible class information can be computed; earlier stages can have precomputed alternative compiled codes that can be selected from once the class hierarchy is known.
- In the face of dynamic class loading, the run-time compilation stage can complete class-analysis-based optimizations based on the knowledge of the program's class hierarchy at a given moment. If future dynamic class loading extends the program's class hierarchy, the dependency and invalidation mechanisms described above can be used to trigger reexecution of the run-time compilation stage.

  Staged class analysis promises to combine highly optimized code with low run-time cost, greatly exceeding the capabilities of systems that have been developed to date.

- Our target applications also are very data-intensive. In high-level object-oriented languages, objects are typically represented as records of pointers to other objects. This standard layout can be very inefficient for large collections of similar objects, in terms of overall space consumption, in cache utilization, and in memory dereference overhead. In separate work we are developing techniques for optimizing the layout of data structures. As one example, a component object could be represented in-line within its containing object rather than indirectly via a pointer, which eliminates pointer dereference costs and space overhead for the pointer and the component's object header. But layout decisions can depend on interprocedural or global properties of the program, or even on the current behavior of the running system. This is exactly the situation where staging can be most effective, and we will investigate staging these layout optimizations, with early stages precomputing possible layouts and the corresponding access operations, and later stages quickly selecting from this palette of choices and patching precompiled code as needed.
- Some of our target applications can make use of parallel processing. As time permits, we will investigate staging appropriate parallel optimizations.
- Similarly, some of our target applications are distributed, dynamically evolving systems, and we will study staging can be applied to these kinds of system.

All of these experiments depend critically on staging being easy to apply to a new optimization, and the results of staging being controllable. Staging these optimizations will thus exploit the features described above in section 3.2 and will provide a good test for their effectiveness.

3.4 Interactions between Run-time Optimization and the Target Architectures

REVISE AS NEEDED/DESIRED (this is the text of the old NGS proposal)

DyC's staged run-time optimizations were designed and applied with relatively little consideration of the underlying target architecture, and all experiments were carried out on single computer, Digital 500 personal workstations, with Alpha 21164 CPUs. However, our preliminary studies and those of other researchers [Lipasti et al. 96, Keppel 96] indicate that different target processor architectures reap different benefits from run-time optimizations. The objectives of our future architectural research are to understand the effect of both uniprocessor and parallel designs on run-time optimization performance, to develop staged optimizations that take the underlying architecture into account, and to apply innovative hardware techniques to improve run-time performance. For these studies we will use the DCPI [Anderson et al. 97], ATOM [Srivastava & Eustace 94], and Kimera [Sirer et al. 98] profiling and binary re-writing tools and the simultaneous multithreading[1] simulation platform (which currently simulates superscalars and multiprocessors, as well as simultaneous multithreaded processors) to determine the amount and cause of processor stalls and issue width utilization. Our experiments will compare bottom-line and component performance for applications executing with and without a particular microarchitectural feature, and with multiple configurations and/or designs of a feature. We will also compare all dynamically generated applications to their statically generated counterparts.

- We will investigate the interaction between specific run-time optimizations and specific microarchitectural features to determine how and why the features affect run-time compilation effectiveness. For example:
  - Run-time compilation both shrinks (by eliminating instructions) and enlarges (by completely unrolling loops and generating multiple specialized versions) the instruction cache footprint. We will investigate the trade-off between instruction cache size and the aggressiveness of the optimizations in order to apply the optimizations appropriately in a given cache hierarchy, and will also determine the role of instruction placement in reducing instruction-cache thrashing.
  - In order to produce benefits on wide-issue processors, run-time optimizations must eliminate instructions on the critical path (eliminating non-critical-path instructions simply frees issue slots). We will investigate techniques for producing increased run-time optimization performance on wide-issue processors, focusing on optimizations that eliminate critical-path instructions and utilize empty issue slots.
  - We will investigate the role of dynamic (hardware) scheduling in improving dynamically generated code that has been simply and cheaply generated in order to reduce run-time optimization overhead.
  - Key optimizations should be sensitive to the implementation of particular instructions. At issue here is not simply the latency of the instructions, but whether hardware implementations for them exist at all. For example, run-time strength reduction is presumably more profitable where integer divides must be converted to floating point operations in statically generated code (as is done on the Alpha 21164).

  These studies should enable better run-time optimization performance and modeling on both uniprocessors and either homogeneous or heterogeneous processors on multiprocessor and distributed systems.
- Parallel and distributed computers change the environment in which complete loop unrolling can be applied. First, the large system-wide instruction caches of multiprocessors and distributed processors should enable complete unrolling of loops that are too large for uniprocessor caches (as examples, the loop that interprets display lists in OpenGL and loops in sparse matrix applications). Second, data distribution [Anderson & Lam 93] and tiling [Lam et al. 91, Carr & Kennedy 92, Coleman & McKinley 95] algorithms, which partition loop nests, should be applied differently in a shared memory multiprocessor (with separate processor data caches) and a simultaneous multithreaded processor (with a thread-shared data cache) [Lo et al. 97b]. With these factors in mind, we will investigate the interaction between complete loop unrolling and parallel/distributed and multithreaded architectures, focusing on such features as the number of processors/threads, the inter-processor and inter-thread communication latencies, and the size and structure of the cache hierarchy, to determine how and when complete loop unrolling can be applied across multiple processors or hardware contexts in a simultaneous multithreaded processor. We will also develop algorithms for dynamically unrolling loops in multiprocessor and simultaneous multithreaded environments.
- Based on the effect of the (uniprocessor and parallel processor) architectural features on run-time compilation performance, we will extend the run-time information around which we apply staged optimizations to include knowledge of the underlying microarchitecture, as well as data values. This will enable our staging infrastructure to take the target architecture (be it local or remote) into account

[1] See Eggers' prior NSF-sponsored research for a description of simultaneous multithreading.

when deciding whether and how to apply run-time optimizations, so that optimizations that are inappropriate for the architecture can be disabled or modified. The architectural information will also comprise parameters in the cost/benefit model for automatic staged compilation.

- Hardware techniques, such as reuse buffers (which store results of instructions that have constant operands and therefore don't need to be reexecuted) [Sodani & Sohi 97] and value prediction tables (which predict instruction results) [Lipasti et al. 96], are alternative approaches to exploiting value consistency and reuse. We will compare run-time compilation to these hardware techniques (1) to uncover the patterns of values that are best matched to each technique and determine which technique performs better in practice, (2) to gauge the relative benefits of microarchitectural features (for example, issue width) on them, and (3) to determine how all techniques can be used together to enhance program performance. Of particular interest is a comparison of reuse buffers and dynamic memoization (in which functions with static variable operands and no side effects are dynamically evaluated once, and the returned result is repeatedly used by the calling procedure).
- Tightly coupled parallel architectures, such as simultaneous multithreading processors (SMT) or single-chip multiprocessors (CMP) [Olukotun et al. 96], provide an opportunity to improve run-time compilation performance by executing in parallel the application to be dynamically compiled and software that facilitates run-time optimization. For example, executing the run-time compilation stage together with an as yet un(run-time)optimized application should largely eliminate the execution-time overhead of run-time compilation. In addition to making run-time compilation essentially free, it will enable more sophisticated optimizations with greater run-time overhead, optimizations with less payoff, and optimizations around variables loaded from memory, whose values are often hard to predict. Alternately, profiling an executing application for value locality can aid in making better decisions about which variables and code to dynamically compile. Using SMT as our target architecture, we will develop the software and hardware support to allow these co-routines to execute and interact seamlessly (such as fork, join, and synchronization primitives and mechanisms for handling thread-shared state) and gauge the performance consequence of executing them in parallel.

4 Target Applications

Early staged compilers have tackled only small programs. We propose to investigate how well staging can be applied to several large (up to hundreds of thousands of lines), complex, long-running, object-oriented applications. We have identified several systems in particular that appear to be good test-beds for our research:

- *UrbanSim* (http://www.urbansim.org) is a software-based simulation model designed to be used for integrated planning and analysis of urban development, incorporating the interactions between land use, transportation, and public policy. It is intended for use by Metropolitan Planning Organizations and others needing to interface existing travel models with new land use forecasting and analysis capabilities. The simulation engine is 300,000 lines of Java source code, making it one of the largest freely available Java programs today. UrbanSim requires a modular software organization in order to be easily extensible with new kinds of simulation agents. It also demands high performance; users would like to perform day-long simulations involving millions of objects interacting at a fine grain. Parallel processing could be useful in enabling larger simulations. Unfortunately, the limitations of current compilation technology have hindered them in combining modularity with high performance, and some parts of their software have had to be rewritten in low-level, unmaintainable ways in order to get acceptable performance. Staged compilation is a promising approach to supporting UrbanSim's modular structure and its performance demands. UrbanSim is an exemplar of the kind of large, complex, performance-intensive server software that is becoming increasingly important.
- *one.world* (http://one.cs.washington.edu) is a system architecture for pervasive computing, where applications must constantly adapt to a highly dynamic computing environment. It provides an

integrated and comprehensive framework for building pervasive applications. It includes a set of services, such as service discovery, checkpointing, and migration, that help to structure applications and directly simplify the task of coping with constant change. One.world is a good example of a program that requires run-time adaptability, which we hope to support effectively with staged compilation. It also is representative of a broad class of distributed middleware platforms.

- *Labscape* (http://labscape.cs.washington.edu) is a ubiquitous laboratory assistant that has the potential to simplify laboratory work, foster new forms of collaboration, and promote the application of computational tools, while helping to train biologists in the skills and standards that will be needed in the laboratories of the future. Labscape is a software and computer hardware system that is always at the biologist's fingertips, anticipates what will happen next in the lab, gathers and organizes data on the fly, and is flexible enough to support the dynamic research process. Like one.world, Labscape is a distributed, dynamically evolving system that requires a high degree of run-time adaptability. It also makes heavy use of dynamic class loading.
- In the longer term, we would like to work with scientists and engineers on our campus to try to apply staged compilation to their problems. The *ZPL* group (http://www.cs.washington.edu/research/zpl) has had good success in transferring their parallel programming language technology to outside users, and we plan to explore ways of applying staged compilation to ZPL programs or to Java versions of such programs.

We have local access to the developers of these programs, enabling us to learn about the key issues in making such systems work efficiently and to get good feedback from potential users of staged compilation.

5 Impact

Staged compilation offers the potential to break free of the constraints of purely static and purely dynamic compilation. By spreading compilation effort across multiple stages, with each stage contributing incrementally to the final optimized solution, staged compilation promises to produce highly optimized target machine code at low run-time cost for highly modular, adaptable applications. The work we propose tackles the obstacles we see to delivering on this promise, by making staged compilers much easier to construct, more optimized, more compact, and more controllable.

This project will improve human resources as well. In addition to the graduate students who will learn how to do research as part of this work on this project, we have had a good history of including undergraduates in our research projects. The infrastructure's support for writing optimization pipelines can be used in graduate and undergraduate compilers classes to make it easier for students to write and experiment with their own optimizations. Staging can be used to give students experience using run-time compilation and help them better understand the trade-offs between static and run-time compilation.

We feel that our proposed research is appropriate for the Next Generation Software program in several respects. Our staged model of computation distributes the entire compilation process across multiple optimization environments (separate compilation time, library assembly time, link time, and most importantly run time), and therefore provides an opportunity to optimize modular, object-oriented programs without incurring high overheads. Individual optimizations are also staged, which improves the effectiveness of traditional optimizations, provides an opportunity to devise new optimizations, and allows optimization to be reactive to a changing hardware and software environment, without incurring high run-time cost. Our staged compiler infrastructure takes into account key features of the target (uniprocessor and parallel) architecture when deciding how to optimize programs; doing so will generate more efficient code for a particular implementation, while maintaining portability across implementations (thereby providing efficient compilation for a heterogeneous environment), and will allow optimizations to be tailored to a changing problem size and hardware resources. Our compilation infrastructure includes the use of profiling information to drive an staging cost/benefit model, which will provide program-, data-, and architecture-specific optimization automatically. Finally, all these factors, taken together, will enable us to more effectively compile applications that traditionally have been slow, improving both their uniprocessor and cross-processor performance.

6 Milestones

This section discusses our planned schedule for completing the proposed research.

Year 1

In the first year, we will focus on getting the SCF staged compilation infrastructure ready for application to some of our target programs. We will develop techniques for better specialization, for manipulating only the unknown parts of the input program in the specialized optimization pipelines, for pre-executing the specialized pipelines, and for trading away minor optimization opportunities to get much faster and smaller specialized pipelines (all described in the first part of section 3.2). We will write additional optimization passes to stage, including target-machine-dependent optimizations and class analysis (described in section 3.3). We will start exercising our staged compiler on the UrbanSim application and either one.world or Labscape.

revise this architecture work and integrate into above paragraph: We will use various profiling tools and the SMT simulator to investigate the interaction between specific run-time optimizations and specific uniprocessor microarchitectural features to determine how and why the features affect staged-compilation performance, and incorporate architecture criteria into the staged optimization algorithms.

Year 2

In the second year, we will seek to complete the basic infrastructure for staged compilation. We will further enhance the staging framework by adding techniques for producing a specialized program representation rather than a specialized optimization pipeline, for giving optimizations the ability to react to how they are being staged, for providing profile feedback to earlier stages, and for managing a dependency framework that will trigger reexecution of a compilation stage if its input information changes (all described in the middle of section 3.2). We will implement some data layout optimizations, and do experiments to see how well staging can be applied to these optimizations. We will include staging within a full compilation infrastructure, incorporating multiple static compilation stages plus at least one run-time compilation stage. We will continue to study our two applications from the first year and also start working with the third application. We will seek additional applications from the scientific and engineering domains.

revise this architecture work and integrate into above paragraph: We will identify the patterns of values for which software-controlled run-time optimizations, hardware reuse buffers, and hardware value prediction caches are most appropriate, and determine whether there are synergistic performance effects in using all techniques together.

We may try to release our staged compilation infrastructure to "friendly" outside researchers, to get some initial feedback on how staging can be applied to other optimizations and other applications.

Year 3

In the final year, we will strive to complete our staged compilation infrastructure. We will extend the core staging algorithms to include feedback from one static stage to earlier stages, including a general mechanism for feeding back both guarantees and hints from both static and run-time stages to earlier stages and for using both invalidation and guard tests to determine when a stage needs to be reexecuted. We will include Calpa-style techniques for automatically determining where staging is most profitable. We will implement tools that allow compiler developers and possibly compiler users to better understand the effects of staging on the way optimizations get specialized. We will consider staging optimizations aimed at parallel and/or distributed programs. We will finish our experiments with our three main applications, and we hope to do additional experiments on other important applications identified in the second year.

revise this architecture work and integrate into above paragraph: We also will study exploiting specialized hardware features, such as separate hardware contexts in an SMT processor, to make run-time profiling and run-time compilation more efficient.

We will make our system freely available to all non-commercial users.

7 Management Plan

Chambers and Eggers will share project management and student advising for most of the research in this proposal, but will divide the responsibilities that fall within their individual areas of technical expertise. In particular, Chambers will provide more advising to students working in staging object-oriented optimizations and Eggers to those working on the architectural issues. Chambers and the project administrator, Melody Kadenko-Ludwa, will be the contacts for NSF administrative and budgetary matters.

8 Summary of Results from Last Five Years of Prior NSF Support

8.1 Craig Chambers

In the last five years, the following NSF grants have supported Chambers's and students' work:

grant #	title	duration	amount
CCR-9457767	National Young Investigator Award	1994-1999	$312,500
CCR-9503741	Analysis and Optimization of Object-Oriented Languages	1995-1998	$149,406
CCR-9970986	Towards Practical, Expressive Multimethods	1999-2002	$297,620
CCR-9975057	A Staged Compilation Architecture for Program Optimization	1999-2001	$500,000
CCR-0073379	Representation Specification and Optimization of Object-Oriented Languages	2000-2003	$249,782

Vortex: Optimizing Compilation for Object-Oriented Languages

The first three grants have supported work on the Vortex optimizing compiler project. The Vortex project has been very successful, producing new technical results in automatic inlining [Dean & Chambers 94], selective recompilation [Chambers et al. 95], link-time class hierarchy analysis [Dean et al. 95b], profile- and class-hierarchy-guided selective procedure specialization [Dean et al. 94, Dean et al. 95a], profile-guided class prediction [Grove et al. 95], interprocedural class analysis frameworks and fast interprocedural class analysis algorithms [Grove et al. 97, DeFouw et al. 98, Grove & Chambers], efficient implementation of advanced method dispatching schemes [Chambers & Chen 99], frameworks for intra- and interprocedural dataflow analysis [Chambers et al. 96], elimination of unnecessary synchronization in Java programs [Aldrich et al. 99, Aldrich et al.], and measurements of optimization effectiveness across languages [Dean et al. 96]. A common theme in these works is assessment of the effectiveness of the optimization techniques on substantial, realistic benchmarks, typically on the order of 10,000-50,000 lines in size, which often are an order of magnitude bigger and more complex than benchmarks studied by previous work. Overall, the techniques studied in Vortex yield speed-ups for the large benchmark programs of 30-40% for C++, of 50-80% for Java (over straightforward translation to C and subsequent optimized compilation of the C code, which is a factor of 4 better than interpreted Java and a factor of 2 better than typical Java “just-in-time” dynamic compilers), and a factor of more than 7 for purely object-oriented Cecil programs. The Vortex compiler system has been in daily use internally for its own development since early 1994, helping to ensure that the techniques used in Vortex are practical on a normal program development environment. The entire Vortex system, including front-ends for C++, Java, and Cecil, and back-ends for portable C code and SPARC assembly code, was originally released to the outside research community in April 1997; a second major public release took place in February 1999. Additional information about Vortex appears in section 2.2.

Many students received training in compiler research as part of the Vortex project. Jeffrey Dean received his Ph.D. in December 1996, and David Grove received his Ph.D. in October 1998. Charles Garrett received his Master’s degree in March 1994. Several other students are early in their Ph.D. work on the Vortex project. Several undergraduates worked as researchers on the Vortex project, including Greg DeFouw (now at Stanford), Tina Wong (now at U.C. Berkeley), Ben Teitelbaum (later at U. Wisconsin), MaryAnn Joy, Brian O’Neill, Leo Shum, and Matt Parker. Vortex also has been used as the infrastructure for implementation projects in my graduate compilers class for several years, aiding the education of dozens of graduate students. Vortex has also been used an an experimental infrastructure for research groups outside the University of Washington.

Whirlwind: A Next-Generation Optimizing Compiler for Object-Oriented Languages

The last two grants listed above supported Chambers’s and students’ research on the new Whirlwind compiler project. Whirlwind builds on the lessons and experience gained with the Vortex project. One major feature in Whirlwind is the use of a sparse intermediate representation, and a framework for automatically composing modular analyses and transformations to achieve better optimization results [Lerner et al. 02]. Another major research direction is the development of techniques for describing the layouts of data structures, for transforming the layouts into more optimized forms [Dolby 97, Dolby & Chien 98, Dolby & Chien 00], and for checking that compiler optimizations preserve important invariants of the code that manipulates these data structures. Two Ph.D. students are currently working on the Whirlwind project.

Cecil: A Purely Object-Oriented Language
MultiJava: An Extension of Java with Open Classes and Multimethods

The first and third NSF grants listed above also supported Chambers’s and students’ research on the Cecil language design project and follow-on language design work. Cecil is a high-level, purely object-oriented language [Chambers 93a, Chambers 96], used as a vehicle to investigate language design issues in classless object models, multi-methods, modules and encapsulation, and static type systems. New technical results include reconciling multi-methods with object-oriented notions [Chambers 92], predicate classes [Chambers 93b], efficient type systems and module systems for languages with multi-methods [Chambers & Leavens 94, Chambers & Leavens 95], modular type checking for languages with multi-methods and open classes [Millstein & Chambers 98, Millstein & Chambers 99, Clifton et al. 00], flexible polymorphic static type systems [Litvinov & Chambers 98, Litvinov 98], a general dispatching model unifying and extending multi-methods, predicate classes, and pattern-matching [Ernst et al. 98]. Vortex is written entirely in Cecil, providing a realistic test-case for evaluating the language. The Cecil front-end was released to the external research community along with the rest of Vortex in April 1997 and then again (along with a stand-alone Cecil interpreter) in February 1999.

We have also built on the Cecil work by adapting the essential ideas to other, more-mainstream languages. For example, the MultiJava language extends Java with open classes (to which new methods can be added from outside of the class definition) and multi-methods, while retaining its modular typechecking.and compilation strategy [Clifton et al. 00]. A prototype MultiJava compiler was released to the outside research community in summer 2001.

Vassily Litvinov is nearing completion of his Ph.D., having worked on both the Vortex and the Cecil projects, and Todd Millstein is closing in fast behind, having worked on the Cecil and MultiJava projects. Several other students are early in their Ph.D. work on the Cecil project. Claudia Chiang, Stuart Williams, Jean Kaiser, and Lauren Bricker received Master's degrees based on early work on the Cecil project. Several undergraduates worked as researchers on the Cecil project, including Vitaly Shmatikov (now at Stanford) and Phiem Huynh Ngoc. Two undergraduates are currently working on the MultiJava project.

DyC: Staged Run-time Compilation for C
Calpa: Automatic Staged Run-time Compilation for C
SCF: A Staged Compilation Framework

The first and fourth grants listed above also supported Chambers's and students' work on the DyC run-time compilation system for C [Auslander et al. 96, Grant et al. 97, Grant et al. 99, Grant et al. 00a, Grant et al. 00b]. The DyC project has developed effective, practical techniques for performing selective run-time compilation on standard imperative programs, driven by declarative and expressive user annotations. A key contribution of the DyC project is its approach to staging several run-time optimizations, with static preplanning drastically reducing the cost of the run-time compilation stage. DyC has been evaluated on larger benchmarks than related research projects. These studies showed that dynamic compilation could produce speedups by generating code that was of sufficient quality to pay back the costs of compiling during program execution, and that the payoff occurred at input sizes that were within the norm for the particular application.

Two current projects are building on our experience with DyC. The Calpa project strives to automate the selection of where and on what to dynamically compile, combining static analysis and profile feedback with a cost/benefit model of DyC's optimizations and the target machine [Mock et al. 99, Mock et al. 00]. The SCF project has built an initial staged compilation framework that can automatically convert a regular unstaged pipeline of optimizations into a staged one, given information about the range of programs to be compiled in the later stages. These two projects were directly supported by our previous grant through the NGS program.

Additional information about DyC, Calpa, and SCF appears in sections 2.3, 2.4, and 3.1.

Brian Grant received his Ph.D. based on his work on the DyC project. Markus Mock is nearing completion of his Ph.D. based on his DyC and Calpa work, and Matthai Philipose is also nearing completion of his Ph.D. based on his DyC and SCF work. All three were co-supervised by the PIs.

8.2 Susan Eggers

NEEDS UPDATING

Presidential Young Investigator Award, Grant No. MIP-9058439, $125,000 in base grants and $185,000 in matching funds, 6/15/90 - 1/31/96

The PYI served as seed funding for research projects in dynamic compilation and locality analysis of shared data. It also funded several projects in entirety; these included a protection model and implementation for single address space architectures, a cost-based code generator that integrates register allocation and code scheduling, the first inline trace generator for shared-memory parallel programs, and a code scheduler that reduces processor stalls caused by memory latency.

Dynamic Compilation

Dynamic compilation optimizes and generates performance-critical code during program execution rather than at the more traditional static compile time. This grant supported the initial feasibility studies that demonstrated that a dynamic compilation system was worth developing [Keppel 96]. In particular, the studies showed that: (1) dynamic compilation could produce speedups by generating code that was of sufficient quality to pay back the costs of compiling during program execution and (2) the payoff occurred at input sizes that were within the norm for the particular application, and (3) speedups were realizable, but quite different, on three architectural platforms (Alpha, MIPS, SPARC). The research that followed was partly funded by the PYI and is described in section 2.1 of this proposal.

Locality Analysis of Shared Data

A sizable portion of interconnect traffic on shared memory multiprocessors is caused by false sharing among processors. False sharing occurs when multiple processors access *different* words within in the same cache block. It is caused by a mismatch between the memory layout of write-shared data and the cross-processor memory reference pattern to it. By restructuring the data to match the pattern, false sharing can be eliminated.

We developed a series of compiler algorithms and a suite of shared data transformations that reduce false sharing [Jeremiassen & Eggers 92, Jeremiassen & Eggers 94]. The algorithms analyze coarse-grained, explicitly parallel programs, determining which data structures are susceptible to false sharing, where processor locality may be improved, and whether and how to transform the shared data at compile-time. Applying these algorithms and transformations improved program execution in several respects [Jeremiassen & Eggers 95]: (1) False sharing was reduced by an average of 66%, and for some programs over 90%. (2) The restructured programs executed up to three times faster than their untransformed counterparts on a KSR-2 multiprocessor. (3) In addition to faster execution times the restructured programs had considerably better scalability (better performance with increasing numbers of processors). (4) When compared to programmer efforts to restructure shared data (including the extensively hand-tuned SPLASH and SPLASH2 benchmarks [Singh et al. 91, Singh et al. 93, Singh et al. 94], our compiler-directed approach never did worse, and for most programs did considerably better.

The PYI-funded research comprised a series of feasibility studies that established that false sharing was the major hindrance to better performance in explicitly parallel programs executing on shared-memory multiprocessors [Eggers & Jeremiassen 91], and showed that programmer-applied data transformations could eliminate it. This led to developing the compiler algorithms that could transform the data automatically, described above.

Integrated Register Allocation and Code Scheduling

Register allocation and code scheduling both utilize registers, but do so independently and often at cross purposes. The major (still!) unresolved issue is whether the lack of communication between them produces inefficient schedules, and, if so, to what extent they must be integrated to improve the generated code.

Our research examined the tension between register allocation and code scheduling over register usage. We developed a cost-based code generator that completely integrates register allocation and code scheduling [Bradlee et al. 91c] and compared it to more traditional approaches. Our results showed that combining register allocation and code scheduling produces more efficient schedules, particularly on programs with very large basic blocks [Bradlee et al. 91b]. We also studied the effects of code generation strategies on several architectural parameters, such as register set size and organization, operation latency and load latency. We found, for example, that integrated code generation strategies require half the registers of traditional techniques to achieve comparable performance [Bradlee et al. 91a].

Multiprocessor Trace Generation

We developed the first inline trace generator for shared-memory parallel programs, called MPTrace, [Eggers et al. 90] that dilates program execution by only a factor of 3 (as compared to 10 to 30 for comparable techniques at the time and up to 1000 for trap-based schemes). We used MPTrace both to study the effects of execution dilation on architectural metrics, such as bus utilization [Koldinger et al. 91], and to generate traces for other research in parallel processing [Eggers & Jeremiassen 91, Jeremiassen & Eggers 92, Tullsen & Eggers 93, Tullsen & Eggers 95, Thekkath & Eggers 94b, Thekkath & Eggers 94a, Jeremiassen & Eggers 94, Jeremiassen & Eggers 95]. MPTrace was also used by dozens of other universities and companies.

Balanced Scheduling

We devised an algorithm for calculating load weights, called *balanced scheduling* [Kerns & Eggers 93], that is targeted for nonblocking processors. It bases its calculations on estimates of instruction-level parallelism in the code. Code generated using this method is modestly more profitable (on average, 5%) than code generated using the more traditional, architecture-based approach. Its advantages stem from its ability to (1) better exploit instruction-level parallelism to hide load latencies and (2) distribute the parallelism across loads in order to be more tolerant of memory latency variability. (In addition, it is independent of the memory system implementation.) We combined balanced scheduling with three optimizations that either increase instruction-level parallelism or utilize it more effectively (loop unrolling, trace scheduling, locality analysis). Our results showed that, coupled with the optimizations, balanced scheduled programs execute up to 40% faster. More importantly, its performance advantage relative to traditional list scheduling, increases to 18% [Lo & Eggers 95].

Code Scheduling under Varying Parallel Resource Constraints, Grant No. CCR-9114167 (Career Advancement Award), $49,999, 16/9/92 - 15/9/93

The research described immediately above was also funded by this grant.

Simultaneous Multithreading, Co-PI with Henry M. Levy, Grant No. MIP-9632977, $807,790, 9/15/96 - 8/31/99

Current wide-issue processors are very underutilized. Experiments we did several years ago showed that a processor capable of executing 8 instructions per cycle was in practice only executing 1.5 to 2. Addressing a single cause of this low utilization, for example, eliminating branch or cache latencies, was not sufficient to rectify the problem, because the culprit was a general lack of instruction-level parallelism (ILP) in the single executing thread. Therefore a general solution was required.

Simultaneous multithreading (SMT) is one such solution. SMT significantly increases processor utilization by issuing and executing instructions from multiple threads each cycle. In particular, if one thread has insufficient instruction-level parallelism to issue 8 instructions, the processor can use instructions from other threads to make up the difference, thus using thread-level parallelism to compensate for low ILP. The performance consequence is significantly higher instruction throughput and program speedups on a variety of workloads that include commercial databases [Lo et al. 98] and scientific applications [Lo et al. 97a] in both multiprogrammed and parallel environments.

Simultaneous multithreading has already had impact in both the academic and commercial communities. The project has produced numerous papers, one of which was the most recent paper selected for the 25th Anniversary Anthology of the *International Symposium on Computer Architecture*, a competition in which the criteria for acceptance was impact. The SMT project at the University of Washington has spawned other university projects in simultaneous multithreading. Finally, at least three U.S. chip manufacturers are currently designing SMT processors for future generations of their microprocessor line. One of these, Digital Equipment Corp. (now Compaq Corp.) has publicly announced the effort.

Specific research contributions of the project include demonstrating speedups on multiple SMT machine models [Tullsen et al. 95], designing an SMT microarchitecture that met our goals of high multiple-thread throughput, minimal single-thread degradation and an implementation that can be easily built from current wide-issue, out-of-order processor technology [Tullsen et al. 96], devising SMT-compatible compiler optimizations [Lo et al. 97b], and operating systems and compiler support for managing SMT's thread-shared register file [Lo et al. 99], and designing a light-weight, processor-resident thread synchronization mechanism that improves the synchronization performance of current, coarse-grain parallel programs and permits the parallelization of new codes that cannot be parallelized with current synchronization mechanisms [Tullsen et al. 99].

Education and Human Resources Impact

One of my most important accomplishments in the past several years is in student supervision. I take my role as student supervisor, teacher, and mentor seriously, and have been producing students who have been among the most highly sought after at a national level. Several have chosen research paths. Dean Tullsen (1997) took a faculty position at U.C. San Diego (and subsequently won a CAREER award) and Tor Jeremiassen (1995) is at Lucent Technologies. Radhika Thekkath (1994) started her career with a postdoc at Stanford under Profs. John Hennessy, Mark Horowitz and Anoop Gupta. (She opted for industry after her postdoc.)

Others have chosen advanced development careers in industry. Dave Bradlee (1991) is at Microsoft, Eric Koldinger (1991) is at Intel, Dan Kerns (1992) is at Pure, David Keppel (1996) is at Transmeta, and Radhika Thekkath (1994) is now at MIPS, Inc.

Much of the software and designs produced by my NSF-sponsored work is available on the Web or has been transferred to industry.

- The inline trace generator source is available in http://www.cs.washinton.edu/research/arch/mptrace.html.
- The Presto parallel applications are available in http://www.cs.washinton.edu/research/arch/presto-apps.html.
- The SMT simulator has been distributed to several researchers doing work in SMT and related processor architectures.
- Intel Corp., Tera Computers and Colusa (now part of Microsoft) have incorporated a form of balanced scheduling into their production compilers/software fault-tolerant software.
- Several chip manufacturers have taken an active interest in simultaneous multithreading. Hank Levy and I have done joint research and patenting with one of them, Digital Equipment Corporation.

We plan to continue this trend by distributing the staged compiler software described in this proposal.

I participated in the DARPA effort to develop the criteria for an optimized, parallelizing national compiler infrastructure that was joint-funded by NSF and DARPA.

Because of my interest in mentoring younger women, I have participated in several CRA-sponsored academic career workshops and other mentoring panels for women in computer science. I will continue this activity at the upcoming FRCC meeting in May.