High-Level Adaptive Program Optimization with ADAPT *

Michael J. Voss and Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN

ABSTRACT

Compile-time optimization is often limited by a lack of target machine and input data set knowledge. Without this information, compilers may be forced to make conservative assumptions to preserve correctness and to avoid performance degradation. In order to cope with this lack of information at compile-time, adaptive and dynamic systems can be used to perform optimization at runtime when complete knowledge of input and machine parameters is available. This paper presents a compiler-supported high-level adaptive optimization system. Users describe, in a domain specific language, optimizations performed by stand-alone optimization tools and backend compiler flags, as well as heuristics for applying these optimizations dynamically at runtime. The ADAPT compiler reads these descriptions and generates application-specific runtime systems to apply the heuristics. To facilitate the usage of existing tools and compilers, overheads are minimized by decoupling optimization from execution. Our system, ADAPT, supports a range of paradigms proposed recently, including dynamic compilation, parameterization and runtime sampling. We demonstrate our system by applying several optimization techniques to a suite of benchmarks on two target machines. ADAPT is shown to consistently outperform statically generated executables, improving performance by as much as 70%.

1. INTRODUCTION

Making accurate compile-time predictions of program performance, and the impact of optimizations on this performance, has always been difficult. Analytical models applied at compile-time must make assumptions that may often be sensitive to input that is unknown until runtime. The same program may have markedly different characteristics when run with different input data sets. Compiler writers are aware of these variations in behavior, and will often choose to not apply a technique if there is the potential that it may degrade performance.

Compounding these challenges, technologies are now converging, with there being only a few dominant processor architectures and operating systems. With this convergence, comes the ability to generate a single executable image for a range of compatible but diverse systems. These compatible systems may differ in many ways, including the configuration and sizes of the memory hierarchy, the network topology and the processor generation and speed. Each of these characteristics can have significant impact on performance and the profitability of optimizations. In addition, emerging technologies such as Condor pools [1] and Grid computing [2, 3] allow users to submit jobs without any advanced knowledge of the exact systems on which they will execute. This lack of machine parameter knowledge only exacerbates the difficulties associated with the static performance prediction and tuning of applications.

To cope with this combined lack of machine parameter and input data set knowledge, recent work has begun to explore adaptive and dynamic optimization paradigms, where optimization is performed at runtime when complete system and input knowledge is available.

1.1 Adaptive Optimization Paradigms

A range of adaptive and dynamic optimization systems have been proposed in the literature. These systems will be discussed in more detail in Section 6, but to highlight the contributions of our framework, we will briefly discuss their major features. Dynamic optimization systems can be broken into three categories: (1) those that choose from statically generated code variants (2) those that modify behavior through parameterization and (3) those that use dynamic compilation.

In approaches like static multiversioning [4] and Dynamic Feedback [5], multiple versions of a code section are generated at compile-time, and at runtime one of these versions is selected based upon runtime values or monitored performance. Using statically generated code variants, these approaches are limited because optimizations are applied before input data set and machine parameter knowledge is available. To guard against the negative impact of *code explosion*, often only a few versions are generated for each code section.

Parameterization attempts to avoid the limitations of static multiversioning. Code is generated at compile-time that can be "restructured" by changing the values of program variables. A simple example is a tiled loop nest where the tile size is a variable. Changing the tile size variable

*This work was supported in part by NSF grants #9703180-CCR and #9975275-EIA.

PPOPP'01, June 18-20, 2001, Snowbird, Utah, USA.

will in effect re-order the loop iterations in different ways with respect to the original untiled loop nest. Gupta and Bodik [6] discuss ways of applying many common transformations through parameterization.

Finally, the most general form of dynamic optimization is dynamic compilation, where code is re-compiled at runtime, allowing compiler optimizations to be applied with complete machine and input data set knowledge available. Dynamic compilation, if loosely interpreted, can include a range of systems from more traditional specializers [7, 8, 9, 10], to binary translators like Dynamo [11] and Java virtual machines like Jalapeno [12].

1.2 A New Approach : ADAPT

In this paper, we proposed a generic compiler-support framework for adaptive program optimization, ADAPT. ADAPT supports dynamic compilation and parameterization paradigms, and like Dynamic Feedback [5], allows users to explore optimization spaces through "runtime sampling".

Using ADAPT language, users specify heuristics for applying optimizations dynamically at runtime. The ADAPT compiler then reads these heuristics and a target application, generating a complete runtime system for applying the user-described techniques. ADAPT removes overheads from the application's critical path by decoupling optimization from execution. The optimizer that is generated by the ADAPT compiler can be run on a free processor of a multiprocessor or, when the target application is being run on a uniprocessor, on a remote machine across the network.

ADAPT, unlike other approaches in the literature, facilitates an iterative modification and search approach to dynamic optimization. Users have available to them the power of dynamic compilation and parameterization, and an efficient framework for runtime sampling. These facilities support iterative modification and search approaches to optimization that (1) generate a range of optimized code versions, which can then be monitored and selected from based on measured performance and (2) dynamically generate new code versions based on the performance of other experimental variants.

The key contribution of this work is to present a dynamic optimization system that:

- is a generic framework, leveraging existing tools,
- can apply diverse optimization techniques,
- understands a domain specific language, AL, with which users can specify adaptive techniques,
- facilitates an iterative modification and search approach to optimization,
- is shown to consistently outperform static optimization approaches.

In the next section, we present an overview of the ADAPT framework. In Section 3, we briefly describe ADAPT Language (AL) and provide a simple example showing its usage. Our experimental setup is described in Section 4 and an evaluation of ADAPT is presented in Section 5. In Section 6, we discuss related work. Section 7 presents our conclusions.

2. AN OVERVIEW OF ADAPT

A dynamic or adaptive optimization system performs three basic functions: (1) it must evaluate the usefulness of applying an optimization technique using current system and data set information, (2) it must be able to apply a program optimization technique if it finds that it will be profitable and (3) it must then be able to re-evaluate its decisions and tune the application as the runtime environment changes. These basic functionalities must be provided in a way that minimizes overheads, so that the benefits of optimization are not offset by the runtime costs.

The ADAPT framework provides these facilities through the approach shown in Figure 1. Code sections that are candidates for optimization have two control paths: (1) a path through a best known version and (2) a path through an experimental version. Each of these versions can be replaced dynamically by the ADAPT optimizer.

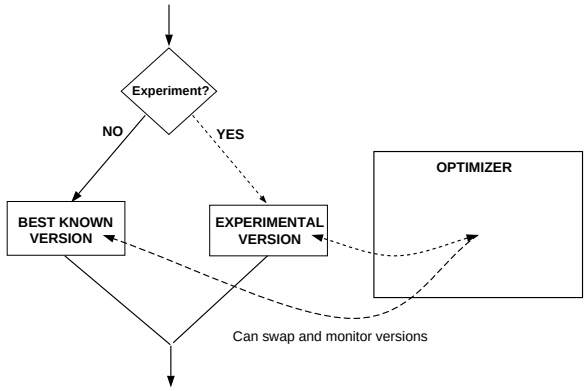


Figure 1: Overview of the ADAPT dynamic optimization system. The shown scheme is applied to each candidate code section in a program.

Before a code section is executed, a flag is checked. If the flag indicates that an experimental version exists, it will be used, otherwise the best known version is executed. The ADAPT optimizer modifies the behavior of the code section by swapping in new best known and experimental versions as the program executes. The experimental version is monitored, and the data collected during its execution can be used as feedback into the optimization process.

The target application's overhead for decision making is small, being only the check of a flag setting and determination of the current context of the interval. ADAPT is able to independently optimize multiple contexts of the same interval, where the context is determined by the value of the loop bounds. All data collection and optimization decisions can be removed from the application's critical path and placed in an optimizer that runs asynchronously in the background. This decoupled structure is the heart of the ADAPT framework, which is pictured in Figure 2.

ADAPT has both a compile-time and runtime component. The ADAPT compiler[1] reads both the target application and user-specified heuristics, and generates a complete runtime system for applying these heuristics to the application

[1]The ADAPT compiler is built on top of the Polaris compiler infrastructure. Polaris is a parallelizing and optimizing source-to-source restructurer [13].

dynamically. The ADAPT compiler first selects interesting code sections (intervals) as candidates for optimizations. Currently, the compiler selects loop nests that contain no I/O and no function calls and that are contained in a cycle in the inter-procedural control flow graph. The compiler then generates a runtime system based on the user heuristics. These heuristics are specified in the domain specific language, ADAPT Language (AL), which is described in more detail in the next section.

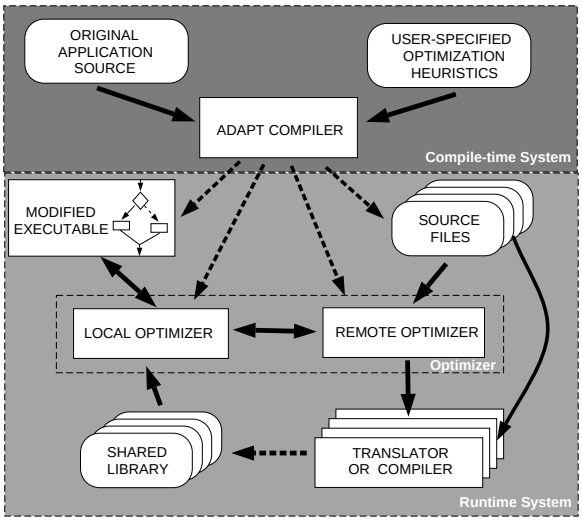


Figure 2: A detailed view of the complete ADAPT framework.

The compiler-generated runtime system consists of a modified version of the application, which contains the two control paths shown in Figure 1 for each candidate interval. It also contains a local optimizer that performs hot-spot detection, determining the most time-consuming code sections in the application as it runs. The local optimizer will communicate with the remote optimizer, and dynamically link in new code variants as they become available. The local optimizer is a separate thread that runs on the same machine as the application. The threading is implemented using posix threads and no locking is required, allowing optimization and execution to truly occur asynchronously.

ADAPT is able to track multiple *contexts* of each interval. Currently these contexts are determined by loop bounds. For example, if a loop in the application sometimes executes with 100 iterations and sometimes with 10,000 iterations, ADAPT will be able to track and optimize these contexts independently of each other. Because of its decoupled structure, ADAPT relies on the repetitive behavior of applications. The optimizer generates new code based on previously seen behavior on a per-context basis. In order for optimization to be profitable, the same context of an interval must be seen multiple times. Decoupling allows optimization of a context to occur concurrently with execution, with the newly optimized code being used whenever it becomes available.

Returning to the framework shown in Figure 2, when a hot-spot is detected, the information about the interval is sent to the remote optimizer using a remote procedure call (RPC). The remote optimizer has available to it source code for each of the candidate intervals, a description of the target machine, and stand-alone tools and compilers that can be used to perform optimizations. The user-supplied heuristics are transformed by the ADAPT compiler into state machines that are applied by the remote optimizer on a per context and per interval basis.

The remote optimizer tunes the program by calling stand-alone tools and optimizers, as well as by selecting new parameters for parameterized code variants. New code variants are stored into shared libraries and provided to the local optimizer via NFS.

When the remote optimizer finishes an optimization step, the RPC invoked by the local optimizer returns, and if a new code variant has been generated, the local optimizer will then dynamically link the code into the target application. This new code variant, depending on the message sent from the remote optimizer, will be swapped in as a new best known version or experimental version for a specific context of an interval. At the next execution of this interval and context, the new code will be available and executed.

In the next section, the ADAPT language is presented, as well as an example of an AL heuristic that will make the operation of the ADAPT framework more concrete.

3. ADAPT LANGUAGE (AL)

ADAPT Language (AL) is a domain specific language with which optimization developers can specify optimization heuristics to be applied to applications dynamically at runtime. It is a C-like language with special statements, some of which are described in Table 1. In addition, it defines reserved words that at runtime contain useful input data set and machine parameter information. For example, a heuristic can directly refer to the L2 cache size of the machine through the use of `l2_size`, or to the number of iterations in an innermost loop by use of `innermost_its`.

AL uses an LL1 grammar, which allows a simple parser to be used. It supports C-style expressions, which are tokenized by the ADAPT compiler but not fully parsed. Using these descriptions and a target application, the ADAPT compiler generates a complete runtime system. A full description of ADAPT Language is beyond the scope of this paper, but in this section we will present an example that illustrates its basic elements.

Figure 3 shows the AL code for defining a *simple_unroller* technique. It begins by declaring variables used in the heuristic, `level` and `min_time`. It then uses *constraint* statements to limit the intervals for which the runtime system will apply this technique. This technique will only be applied to intervals that are perfectly nested loops, and for which the number of iterations in the innermost loop is known at the entry to the interval.

Next, an `apply_spec` statement defines the interface to the tool that will be used to apply the optimization. The first parameter determines when this interface will be used. If a `collect` or `mark_as_best` statement is issued and `level > 1`, then a code variant generated using "-unroll=level" will be created (or looked up, if it already exists). The second parameter in the `apply_spec` statement, `bflag`, identifies that the technique is a flag that is added to a call to the backend compiler. The last parameter defines the interface itself. The flag is "-unroll=level" where, at runtime, level will be replaced by its value.

The `phase` block defines the heuristic, which is transformed by the ADAPT compiler into a state machine used

Table 1: Special ADAPT Language Statements

Statement	Description
constraint(*compile-time constraint*)	Supplies a *compile-time constraint*. Only intervals that meet all specified constraints will be prepared for runtime optimization.
apply_spec(*condition*,*type*,*syntax*[,*params*])	A description of a tool or flag. Specifies the runtime *condition* under which it is applied, the *type* of tool, the *syntax* of the tool interface and the runtime *parameters* that the generated code will need to be passed.
collect (*event_list*) execute;	Initiates the monitoring of an experimental code version. The *event_list* specifies what events are to be measured.
mark_as_best	Specifies that the code variant that would be generated under the current runtime conditions is a new best known version
end_phase	Denotes the end of an optimization phase

```
technique simple_unroller {

  int level;
  float min_time;

  constraint(is_perfect_nest);
  constraint(inner_its_known);

  apply_spec(level > 1, bflag,
             -unroll=level);

  phase {
      level = 1;
      min_time = FLT_MAX;
      collect ( time ) { execute };
      if ( time < min_time ) {
        min_time = time;
        mark_as_best;
      }
      while ( level < innermost_its ) {
          level = level + 1;
          collect ( time ) { execute; }
          if ( time < min_time ) {
            min_time = time;
            mark_as_best;
          }
      }
      end_phase;
  }
}
```

Figure 3: The AL description of a simple dynamic loop unroller.

by the remote optimizer. Essentially, all intervals will be timed without being unrolled, and then timed after being unrolled by all factors up to, and including, complete unrolling of the innermost loop. At each step, the code section is unrolled by `level` and then swapped in as the experimental version.

Whenever a `collect` statement is executed by the state machine, a message is sent from the remote optimizer to the local optimizer that initiates monitoring of a code variant. The application will then, at the next invocation of that interval of the code section, use the experimental variant described by the message. When executed in the application, these variants are timed. At the `collect` statement, the remote optimizer pauses the AL heuristic, knowing that it must wait for the local optimizer to return information about the behavior of the executed version. While it is waiting, the remote optimizer is free to work on other contexts and intervals. When the local optimizer sees that the experimental version has been timed, and if this context of the interval is still important (i.e. a hot-spot), it will again call the remote optimizer, passing it the results of the monitoring.

After each collection point in Figure 3, the heuristic checks if the execution time for the experimental variant is lower than any that has been previously measured for that context of the interval. If indeed it is an improvement, this new variant will be marked as the new best known version. As with the `collect` statement, the `mark_as_best` statement also causes a message to be passed back to the local optimizer, informing it that a new variant should be moved into the *best known* position for that context.

Finally, when the `while` loop finishes, the `end_phase` statement will send a message to the local optimizer stating that this context has been fully optimized by this technique. This message will cause a timestamp to be kept by the local optimizer. Users can specify a time period after which best known versions become stale, causing heuristics to be rerun. When the local optimizer performs hot-spot detection, it ignores all intervals that have been fully optimized, until the time period since their full optimization exceeds this staleness parameter. If the local optimizer determines that a best known version has become stale, the AL heuristics will be restarted to determine a new best version.

Dynamic compilation is implicit in AL heuristics. When the remote optimizer needs to pass a message to the local optimizer, it first determines if the code variant that it describes exists. If not, then the interfaces described by the `apply_spec` statements are invoked to create the new code variant. For example, in Figure 3, a `collect` statement may cause a "-unroll=4" variant to be timed and then the following `mark_as_best` statement may identify it as new best known variant for that context. If this interval had not yet been unrolled by a factor of 4, then the `collect` statement would cause a "-unroll=4" variant to be compiled. Later, when the `mark_as_best` statement is executed, it would see that a variant matching its description already exists, and simply pass its location to the local optimizer. Likewise, if later for a different context of the same interval, a "-unroll=4" variant were needed, this same code variant could again be used.

The "users" targeted by ADAPT Language are compiler writers who wish to explore the possibilities for dynamic program optimization. A compiler writer who experiments with AL and finds that a particular technique may be profitably applied in a dynamic way, may then either continue to use ADAPT as its mechanism of application, or design a special purpose system around the technique.

4. EXPERIMENTAL SETUP

To demonstrate the effectiveness of the ADAPT framework, we will present the results from six experiments run on two different architectures. The machines used for our study are described in Table 2, and include a six processor Sun UltraSPARC Enterprise 4000 (E4000) and a uniprocessor Pentium workstation running Linux. Currently, the ADAPT framework and its compiler have been ported to both Solaris and Linux. When using the E4000, the remote optimizer is run on a free processor of the multiprocessor, and when using the Pentium workstation, the remote optimizer is run on another identical workstation across the network.

Table 2: Target machine configurations.

	Sun E4000	Pentium II Workstation
OS	Solaris 2.6	Red Hat Linux 6.2
# Cpus	6	1
Cpu Type	UltraSPARC II	Pentium II
Cpu Clk	250 MHz	300 MHz
Mem	1 GB	128 MB
L1 Cache	16 KB	16 KB
L2 Cache	1 MB	512 KB
Compiler	SunPro v 5.0	Gcc (egcs-2.91.66)

We evaluate ADAPT by performing the six experiments described in Table 3. The first experiment *Useless Copying* aims at uncovering the overheads inherent in the system. The remaining five experiments highlight the variety of techniques that can be implemented in our framework. The dynamic techniques we evaluate include loop-bound specialization, back-end flag selection, loop unrolling, loop tiling and automatic parallelization.

We apply these techniques and our framework to five programs: (1) Applu, (2) Mgrid, (3) Npow, (4) Pde and (5) Swim. Applu, Mgrid and Swim are SPEC2000 floating point benchmarks. Npow is a kernel that raises a matrix to a power through repeated matrix multiplication. We include this kernel as a reference point since matrix multiplication is often used to evaluate optimization techniques in literature. Pde is a simple finite difference solver. Pde performs updates in place in the grid using the Gauss-Seidel method. The number of lines, number of intervals identified as candidates for optimization, and the execution time of the original code is shown for each benchmark in Table 4.

Table 4: Program Descriptions.

Program	Lines	Intervals	Time (Sun E4000)	Time (Linux)
Applu	3890	32	2736 sec.	1123 sec.
Mgrid	489	13	4185 sec.	575 sec.
Npow	72	5	488 sec.	587 sec.
Pde	36	2	1531 sec.	533 sec.
Swim	429	14	467 sec.	872 sec.

5. EXPERIMENTAL RESULTS

The results of the experiments described in Table 3 are shown in Figure 4. The execution time of each application optimized by ADAPT is shown as a percentage of the execution time of the original, statically optimized code. These times are the wall-clock times for the entire execution of the application. For some of the experiments the performance of code generated by statically applying the same optimization technique is presented. We will discuss each experiment in detail below.

Useless Copying

The Useless Copying technique was designed to uncover the overheads associated with the ADAPT framework. The "optimization" generates an identical program. Since no improvement is to be expected from this transformation, the changes in performance are due to the impact of the framework itself.

Overheads on all applications on both target machines are always less than 5% as shown in Figure 4.a. In many cases, especially on the Pentium II Linux workstation, there is a negative overhead, or improvement, seen by applying ADAPT. The reason for these improvements are due to the restructuring performed by the ADAPT compiler. While transforming the applications and generating the runtime system, the ADAPT compiler performs inter-procedural constant propagation and applies some simplifications that may lead to improved performance.

From Figure 4.a, it is clear that the overheads incurred by the ADAPT infrastructure are minimal. On average, the applications slowdown by 0.2% on the Sun E4000 and improve by 1.6% on the Pentium II Linux workstation.

Loop-Bound Specialization

In loop-bound specialization, the variables that determine loop bounds are replaced in each context by their runtime constant value, and then the code is recompiled by the backend compiler. All improvements are due to the backend compiler exploiting this new input data set knowledge.

While in Figure 4.b there is little improvement in many applications, Npow shows a dramatic improvement of almost 70% on the Sun E4000. When given the loop-bound information for Npow, the Sun compiler performs aggressive loop unrolling and is able to remove many branches by knowing the exact number of iterations executed in the matrix multiply. On average, loop-bound specialization leads to an improvement of 13.6% on the Sun E4000 and 2.2% on the Pentium II Linux workstation.

Back-End Flag Selection

Figure 4.c shows the improvements made when ADAPT was used to select the best collection of backend compiler flags for compiling each interval. The flag choices available vary across the two architectures due to the use of two different compilers.

On the E4000, the base case is compiled using the SunPro compiler with the `-fast` flag. This flag expands into what the compiler writers decided was the best collection of flags for the target machine. These flags provide much machine specific information, including the type of chip and the sizes of the memory hierarchy.

The flag selection experiment uses ADAPT to simply toggle the flags that are implied by the `-fast` switch. For example, `-fast` implies `-prefetch=no`. ADAPT will therefore compare the performance of code sections compiled with both `-prefetch=no` and `-prefetch=yes`, selecting as best the version that shows the shortest execution time.

The `-fast` switch implies eight compiler flags. We defined AL heuristics for each of these flags and combined them using a linear search. On average ADAPT was able to improve

Table 3: Overview of experiments

Experiment	Description
Useless Copying	Six identical copies of each interval are made and the timed. The copy with the smallest execution time will be selected as best. This experiments shows the overheads inherent in the framework, since no improvement should be expected through copying.
Specialization	The variables that determine the loop bounds in each interval are replaced as constants by their runtime values. Any improvement will be due to the backend compilers ability to exploit this added information.
Flag Selection	The application begins with highly optimized code. On Solaris, the -fast compiler flag is used and on Linux the -O2 flag is used. Additional flags available in the backend compilers are then experimented with by doing a linear search on their settings and finding the sets that show the smallest execution time. On Sun we look at 8 flags and on Linux we look at 9 flags.
Loop Unrolling	Loop nests that contain a single innermost loop are unrolled. First the original code is timed. Then the loop is unrolled by factors that evenly divide the number of iterations of the innermost loop to a maximum unroll factor of 10. At each step the unrolled variant is timed. As soon as the execution time of a monitored variant is larger than the best known version, optimization is halted and the best known version is assumed to be best.
Loop Tiling	Loops nests that are tilable, and for which exploitable temporal locality exists will be optimized. First the original code is timed. The the nest is tiled for 1/2 of the L2 cache size and timed. The code is then tiled to exploit $(1/2)^n$ of the cache size down to 1/16th. This attempts to find the *effective* cache size of the machine. The version that shows the lowest execution time is selected as best.
Parallelization	For each loop nest that is determined to be parallel by the Polaris compiler, both a parallel and serial version is executed and timed. The version that shows the lowest execution time for that context will be chosen as best.

overall program performance by 35% due to its runtime selection of flags as shown in Figure 4.c.

Table 5 shows an example of an execution trace for one of the code sections in Mgrid when optimized by ADAPT on the Sun E4000. This table shows the code versions generated and monitored by the AL heuristics in the order that they were executed. The leftmost column provides the code version number and the rightmost column provides the execution time of that variant. The flags with the greatest impact on the code section were prefetching (PF), which when turned on, decreased the execution time by 66%, and floating point simplification (FP) which, when raised to a higher level, improved performance by an additional 19%. Overall, on this context of the interval, the dynamic selection of flags beats the manufacturer's selection by 73%.

Table 5: Flags Trace in Mgrid on Sun.

Ver	PF	XV	O	FP	Dp	Ca	Ch	Tr	XTime
1	N	N	4	1	Y	Y	Y	N	1.05 s
2	Y	N	4	1	Y	Y	Y	N	0.36 s
3	Y	Y	4	1	Y	Y	Y	N	0.36 s
4	Y	Y	5	1	Y	Y	Y	N	0.37 s
5	Y	Y	4	2	Y	Y	Y	N	0.29 s
6	Y	Y	4	2	N	Y	Y	N	0.28 s
7	Y	Y	4	2	N	N	Y	N	0.28 s
8	Y	Y	4	2	N	Y	N	N	0.29 s
9	Y	Y	4	2	N	Y	Y	Y	0.28 s

On the Pentium II Linux workstation, the base versions were compiled with gcc using the `-O2` flag. Gcc also provides several other optimization flags. We selected nine of these flags, including `-O`, and as in the Sun E4000 experiment, performed a linear search of their settings for each context of each interval. On average, ADAPT was able to improve performance over the statically optimized code by 9.2%. An example execution trace for one of the code sections in Swim is shown in Table 6. Interestingly in this example, non-intuitive choices have large impacts. For example, moving from `-O2` down to `-O` improves performance by 23%.

Table 6: Flags Trace in Swim on Linux.

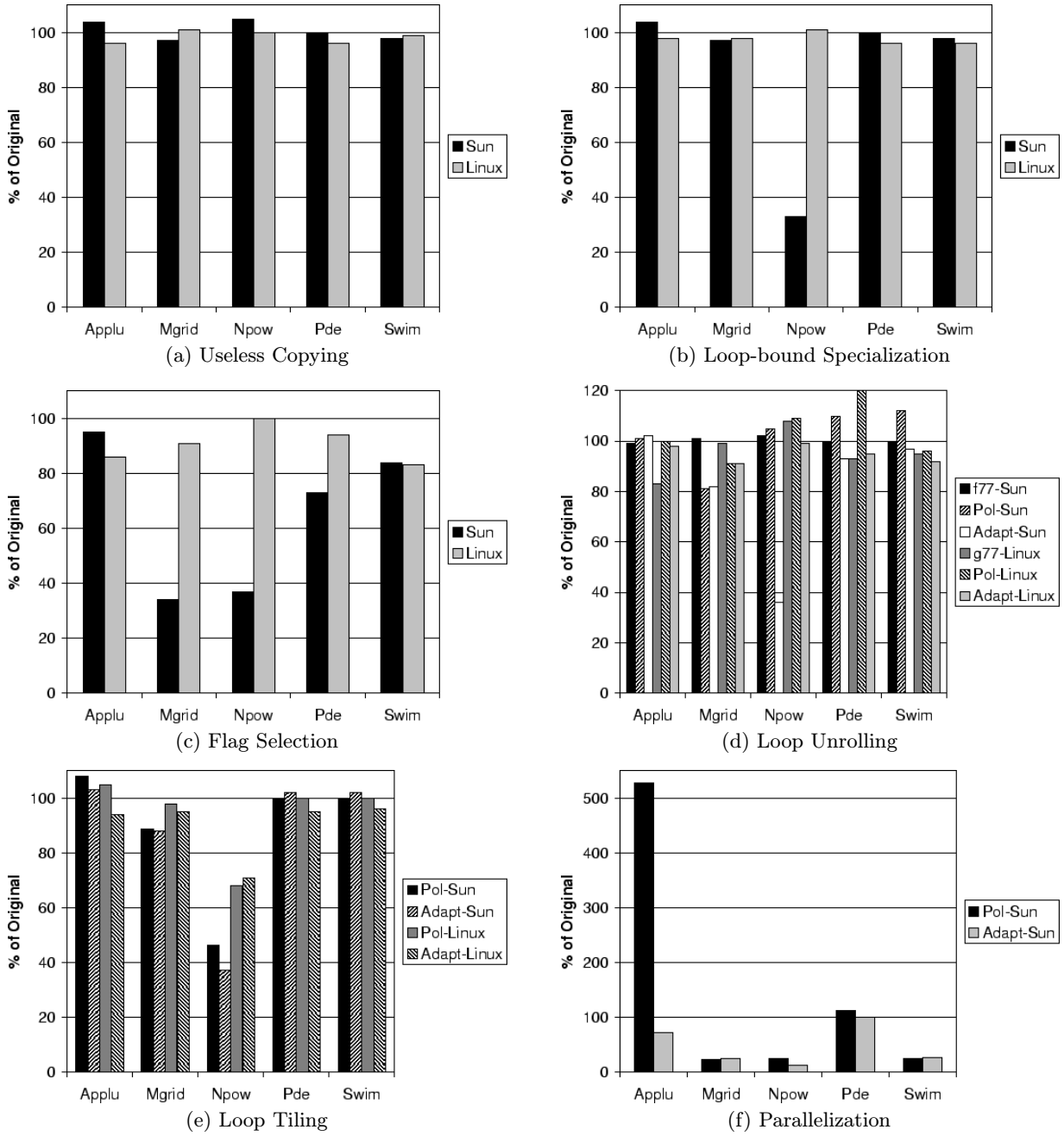
Ver	O	Ur	Ma	Db	FS	Ex	Me	St	CSE	XTime
1	2	N	N	N	N	N	N	N	N	0.158 s
2	0	N	N	N	N	N	N	N	N	0.122 s
3	0	Y	N	N	N	N	N	N	N	0.144 s
4	0	N	Y	N	N	N	N	N	N	0.123 s
5	0	N	N	Y	N	N	N	N	N	0.122 s
6	0	N	N	N	Y	N	N	N	N	0.124 s
7	0	N	N	N	N	Y	N	N	N	0.123 s
8	0	N	N	N	N	N	Y	N	N	0.118 s
9	0	N	N	N	N	N	Y	Y	N	0.162 s
10	0	N	N	N	N	N	Y	N	Y	0.118 s

Loop Unrolling

In Figure 4.d, the dynamic loop unrolling technique implemented using ADAPT is compared to two static approaches. First, on the Sun E4000 the applications are compiled by the Sun compiler with `-unroll=2`, and on the Pentium II Linux workstation, the applications are compiled with gcc using `-funroll_loops`. In Figure 4.d, these application variants are labeled f77-Sun and g77-Linux, respectively. Both the Sun compiler and gcc take these flags as hints, and unroll loops only if their built-in heuristics determine that the unrolling may be profitable.

To force an unrolled static version for comparison, we modified the Polaris compiler to perform loop unrolling. Statically optimized versions were generated for each application using Polaris with `-unroll=2`. The unroll factor of 2 was used for both Polaris and the Sun compiler since this level was most often selected as the best unrolling factor by ADAPT in these benchmark codes.

Figure 4.d shows that only in Applu on the Pentium II Linux workstation is ADAPT not close to, or better than, the best statically generated code variant. This is due to a sheltering mechanism that is included in the runtime system. ADAPT will not optimize intervals that have very short execution times, since it is likely that the framework overheads will dominate. Applu contains small loops, many of which



(a) Useless Copying

(b) Loop-bound Specialization

(c) Flag Selection

(d) Loop Unrolling

(e) Loop Tiling

(f) Parallelization

Figure 4: Experimental results on both the Sun E4000 and the Pentium II Linux Workstation

are sheltered by ADAPT. In Applu, the static compiler is able to also unroll these loops, while the runtime system ignores them to avoid potential degradation.

On the Sun E4000, the backend compiler is able to improve performance through unrolling by an average of -0.4% and Polaris by -1.8%, while ADAPT sees an average improvement of 18%. On the Pentium II Linux workstation, gcc is able to improve performance on average by 4.4%, Polaris by -3.2%, and ADAPT by 5%.

Loop Tiling

In the loop tiling experiment, ADAPT's performance is compared to Polaris when given exact knowledge of the target machine's L2 cache size. Polaris will tile loops to exploit 1/2 of the L2 cache size if either (1) the number of accesses provably exceeds the tile size or (2) the number of iterations, and hence the number of accesses, is unknown at compile-time.

Unlike Polaris, ADAPT has exact knowledge of loop-bound information, and will tile loops only when the number of accesses truly exceed the tile size. ADAPT will try tile sizes ranging from 1/2 the L2 cache size to 1/16 of the L2 cache size.

As shown in Figure 4.e, even without input data set knowledge, Polaris is able to perform nearly as well as ADAPT. On the Sun E4000, Polaris improves performance on average by 13.4% and ADAPT by 13.5%. While on

the Pentium II Linux workstation, Polaris improves performance on average by 5.8% and ADAPT by 9.8%.

Parallelization

The final experiment is automatic parallelization. Since only the E4000 is a multiprocessor, results are not shown for the Pentium workstation. Data is shown in Figure 4.f for both the code as parallelized by Polaris, and for the code as parallelized by ADAPT.

The AL heuristics fed to ADAPT yields a system that times each code section that is found to be parallel by Polaris. It times the intervals both when run in parallel and when run sequentially. It then chooses the variant with the shortest execution time as best. This experiment is not runtime data dependence testing, but instead is a technique for parallelizing only those pieces of the code that have enough work to mitigate the overheads associated with their parallel execution.

It is clear from the results in Figure 4.f, that Polaris chooses unwisely in Applu, degrading performance significantly. By identifying those code sections that show improvements through parallelization, ADAPT not only removes the degradation, but improves the performance of Applu by 25%. A similar removal of overheads is seen in Pde. On average, when run with its default settings, Polaris degrades performance by 41% on the E4000, while ADAPT improves performance by 52.8%.

Summary of Results

The average improvement for the various techniques as applied on the Sun Enterprise are shown in Figure 5.a, and as applied on the Pentium workstation in Figure 5.b. It is clear that on average, ADAPT offers a better solution than the static optimization tools.

In addition, the biggest slowdowns and biggest speedups obtained through unrolling, tiling and parallelization are shown in Table 7. ADAPT always shows the least degradation, and only in two cases does it not also have the largest improvement. ADAPT through its ability to exploit input data set and machine parameters, as well as its ability to verify its choices through runtime sampling, offers the best performance with the least risk for degradation.

6. RELATED WORK

One of the earliest methods proposed for performing runtime optimization was multiple version loops [4]. In this technique, several variants of a loop are generated at compile-time and the best version is selected based on runtime information. Many compilers still employ this technique. As discussed previously, multiversioning can lead to code explosion since it cannot make use of runtime information to specialize the code that is generated.

Gupta and Bodik [6] proposed *adaptive loop transformations* to allow the application of many standard loop transformations at runtime using parameterization. They argue that the applicability and usefulness of many of these transformations cannot be determined at compile-time. Although they do not give criteria for selecting transformations at runtime, they provide a framework for applying loop fusion, loop fission, loop interchange, loop alignment and loop reversal efficiently.

Diniz and Rinard [5] proposed *dynamic feedback*, a technique for dynamically selecting code variants based on measured execution times. In their scheme, a program has alternating sampling and production phases. In the sampling phase, code variants, generated at compile-time using different optimization strategies, are executed and timed. This phase continues for a user-defined interval. After the interval expires, the code variant that exhibited the best execution time is used.

Like dynamic feedback, Saavedra and Park [14] propose *adaptive execution*, which dynamically adapts program execution to changes in program and machine conditions. In addition to execution time, they use performance information collected from hardware monitors.

A dynamic technique often discussed in relation to parallel processing is runtime data dependence testing. In [15, 16, 17], runtime tests are performed to uncover parallelism undetectable at compile-time. The authors discuss *schedule reuse*, a phenomenon which can be exploited to reduce the number of times a test needs to be applied. Work has also been done by Hall and Martonosi [18] to dynamically select the best number of processors to use for a parallel application when run in a multiprogram environment. The work in [18] was aimed at increasing throughput by allowing applications to cooperate, yielding and taking processors according to their parallel behavior.

The approaches discussed above selected from previously generated code, or modified program execution through parameterization. Much work has also been done on dynamic compilation and code generation [19, 20, 21, 7, 8, 9, 10]. This work has primarily focused on efficient runtime generation and specialization of code sections that are identified through user-inserted code or directives. Dynamic compilation usually falls directly in the application's critical path. To reduce the time spent in code generation, optimizations are staged by using compilers that are specialized to the part of the program being optimized [19].

Work has also been done to collect binary program traces and to optimize these traces during program execution. The HP Dynamo project [11] has shown significant results using such a scheme. The traces collected by Dynamo extend beyond basic blocks and subroutine boundaries, allowing the runtime compiler to be less restricted by control flow and even file boundaries. These approaches create larger blocks with simplified control flow, facilitating many traditional compilation techniques. Dynamo operates on the runtime stream of assembly-level instructions and, its overhead being in the critical path, is constrained by runtime overhead. It is therefore primarily constrained to peephole-like optimizations, since at the low-level at which it operates, knowledge of high-level constructs is unavailable. The Dynamo approach can been seen as complementary to ADAPT which is aimed at higher-level techniques.

ADAPT attempts to minimize runtime overhead by removing code generation from the critical path. Therefore it obviates the need for specialized compilers. Plezbert and Cytron [22] have proposed *continuous compilation* to overlap the "just-in-time" compilation of Java applications with their interpretation. Compilation occurs in the background as the program continues to be executed through interpretation. They also order the code section to be compiled by targeting hot-spots first. This is also the approach taken by the Java HotSpot Performance Engine [23]. Unlike our approach, these approaches may make use of machine parameter information, but do not specialize code using input data set knowledge, and provide no feedback mechanisms.

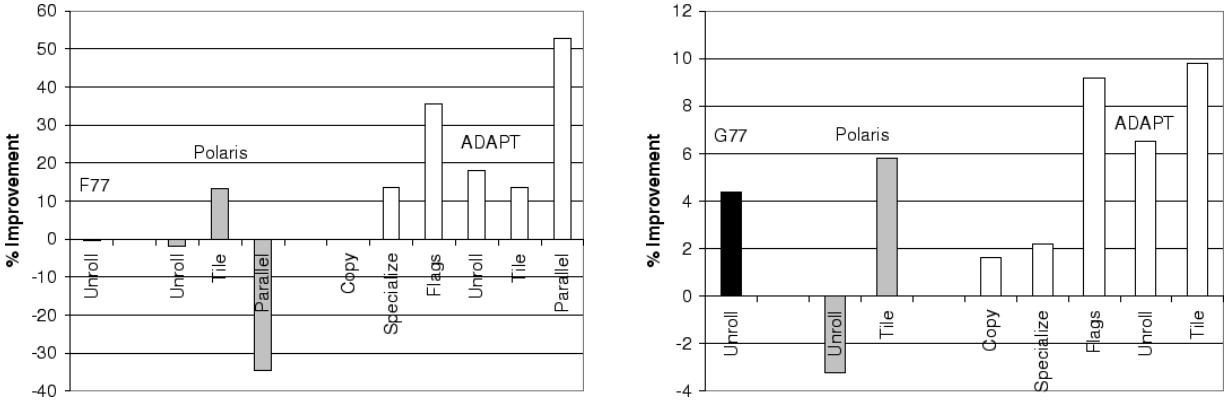


Figure 5: Average performance on (a) the Sun E4000 and (b) the Pentium Workstation.

Table 7: The best and worst performance for Unrolling, Tiling and Parallelization.

	Biggest Slowdown			Biggest Speedup		
	Unroll (Sun,Linux)	Tile (Sun, Linux)	Parallel (Sun)	Unroll (Sun,Linux)	Tile (Sun, Linux)	Parallel (Sun)
Backend	2%, 8%	NA	NA	0.9%, 17%	NA	NA
Polaris	12%, 20%	8%,5%	418%	19%, 9%	53.7%, 32%	76%
ADAPT	1.8%, none	3%,none	1.1%	64%, 9%	62.8%, 29%	87%

The IBM Jalapeno JVM, on the other hand, does provide a feedback mechanism. In [12], they discuss their feedback-directed optimization and show its application to function inlining. While they discuss a general framework, currently only inlining has been evaluated in the literature. Hence, direct comparison with ADAPT is difficult. Like the other Java optimizers, Jalapeno does allow overlap of optimization and execution. However, unlike ADAPT, where optimization and execution is truly asynchronous, the Jalapeno optimizer must hold the master JVM lock during compilation, potentially introducing contention between the compilation and application threads.

In [24], we presented an early version of the ADAPT framework. Our current implementation is different in approach. In [24], ADAPT was not context sensitive, but instead viewed applications as phase oriented. Not being context based, the early version of ADAPT could not support statically unsafe optimizations, such as specialization. In addition, there was also no support for parameterization, which we used in this paper to implement tiling. In [24], addition of new techniques required the writing of a C++ class that was then compiled into the ADAPT compiler. With the AL heuristic language we now support, the types of optimization paradigms that can be implemented are much more diverse.

7. CONCLUSION

With converging technologies that allow applications to be run portably across increasingly diverse systems, adaptive and dynamic program optimization is clearly an important emerging technology. Coping with these new and dynamic environments will be a true challenge for developers of both scientific and mainstream applications. In this paper, we presented ADAPT, a framework that allows researchers to meet these new challenges, to experiment with the adaptive application of both traditional and new techniques, and to develop a better understanding of the options involved in dynamic and adaptive optimization.

ADAPT is a generic compiler-supported framework for high-level adaptive program optimization. Using ADAPT Language (AL), users can easily construct adaptive optimizations by leveraging existing stand-alone optimization tools and compilers. The ADAPT compiler reads user-supplied heuristics and a target application. It then generates a complete runtime system for applying these heuristics dynamically. The heuristic-based approach and the use of a domain-specific language is unique to our framework.

Using a decoupled structure, ADAPT removes optimization overheads from the application's critical path. All optimization decisions can be performed on a free processor if executing on a multiprocessor, or on a remote system when executing on a networked uniprocessor.

ADAPT is applicable to both serial and parallel programs. However, given the many options and the importance of high performance for parallel applications, ADAPT is particularly well suited to these types of applications. Being based on a decoupled approach that overlaps optimization and execution, this system also relies on repetitive behavior, which is often found in parallel science and engineering applications. We evaluated ADAPT by performing six experiments on two target machines. These experiments demonstrated the ability of ADAPT to apply a wide range of techniques effectively on both a multiprocessor UltraSPARC Enterprise server and on a uniprocessor Pentium workstation.

On average, ADAPT was able to significantly outperform static optimization alternatives, showing improvements as large as 70%. In a *flag selection* experiment, it was able to significantly outperform a compiler manufacturers choice of the best collection of compiler flags for the given architecture. It was also able to identify non-intuitive choices that a

user would be unlikely to try. ADAPT showed the least risk for degradation across the various experiments and target machines. Since compiler writers are aware of the potential for slowdown from "optimization", techniques are often applied conservatively, or not all. With ADAPT, degradation is minimized, allowing optimizations to be performed more aggressively, and with accurate input data set and machine parameter knowledge available.

8. REFERENCES

[1] M. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proc. of the 8th Int'l Conf. of Distributed Computing Systems*, pages 104–111, June 1988.

[2] Nirav H. Kapadia and José A.B. Fortes. On the Design of a Demand-Based Network-Computing System: The Purdue University Network Computing Hubs. In *Proc. of IEEE Symposium on High Performance Distributed Computing*, pages 71–80, Chicago, IL, 1998.

[3] Ian Foster and Carl Kesselmann. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, January 1997.

[4] M. Byler, J.R.B. Davies, C. Huson, B. Leasure, and M. Wolfe. Multiple version loops. In *International Conf. on Parallel Processing*, pages 312–318, August 1987.

[5] Pedro Diniz and Matrin Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. of the ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, pages 71–84, Las Vegas, NV, May 1997.

[6] Rajiv Gupta and Rastislav Bodik. Adaptive loop transformations for scientific programs. In *IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, San Antonio, Texas, October 1995.

[7] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proc. of the SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 149–159, Philedelphia, PA, May 1996.

[8] Charles Consel and Francois Noel. A general approach for run-time specialization and its application to C. In *Proc. of the SIGPLAN '96 Conf. on Principles of Programming Languages*, January 1996.

[9] D. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proc. of the SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 160–170, Philedelphia, PA, May 1996.

[10] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proc. of the SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 137–148, Philedelphia, PA, May 1996.

[11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.

[12] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proc. of the ACM SIGPLAN 2000 Conf. on Object-Oriented Programming Systems, Languages and Applications*, Minneapolis, MN, October 2000.

[13] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel Programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.

[14] R. Saavedra and D. Park. Improving the effectiveness of software prefetching with adaptive execution. In *Proc. of the 1996 Conf. on Parallel Algorithms and Compilation Techniques*, Boston, MA, October 1996.

[15] J. Saltz, R. Mirchandaney, and K. Crowley. Run time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[16] Lawrence Rauchwerger and David Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization . *Proceedings of the 8th ACM International Conference on Supercomputing, Manchester, England*, pages 33–43, July 1994.

[17] L. Rauchwerger and D. Padua. The LRPD Test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Languages Design and Implementation*, pages 218–232, June 1995.

[18] Mary W. Hall and Margaret Martonosi. Adaptive parallelism in compiler-parallelized code. In *Proc. of the 2nd SUIF Compiler Workshop*, August 1997.

[19] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. of the SIGPLAN '99 Conf. on Programming Language Design and Implementation*, pages 293–304, Atlanta, GA, May 1999.

[20] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. In *Proc. of the SIGPLAN '99 Conf. on Programming Language Design and Implementation*, pages 281–292, Atlanta, GA, May 1999.

[21] Massimiliano Polettto, Wilson C Hsieh, Dawson R Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.

[22] Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late than never"? In *Proc. of the ACM SIGPLAN-SIGACT '97 Symposium on Principles of Programming Languages*, pages 120–131, Paris, France, January 1997.

[23] Sun Microsystems. The Java HotSpot Performance Engine Architecture. Technical White Paper, http://java.sun.com/products/hotspot/whitepaper.html, April 1999.

[24] Michael Voss and Rudolf Eigenmann. ADAPT: Automated De-Coupled Adaptive Program Transformation. In *Proc. of the International Conf. on Parallel Processing*, Toronto, Ontario, August 2000.