Dynamic Run-time Architecture Techniques for Enabling Continuous Optimization

Tipp Moseley†, Alex Shye, Vijay Janapa Reddi, Matthew Iyer, Dan Fay,
David Hodgdon, Joshua L. Kihm, Alex Settle, Dirk Grunwald†, Daniel A. Connors

Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309-0430
{shye,janapare,iyer,faydr,hodgdon,kihm,mw.settle,dconnors}@colorado.edu

†Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
{moseleyt,grunwald}@colorado.edu

ABSTRACT

Future computer systems will integrate tens of multithreaded processor cores on a single chip die, resulting in hundreds of concurrent program threads sharing system resources. These designs will be the cornerstone of improving throughput in high-performance computing and server environments. However, to date, appropriate systems software (operating system, run-time system, and compiler) technologies for these emerging machines have not been adequately explored. Future processors will require sophisticated hardware monitoring units to continuously feed back resource utilization information to allow the operating system to make optimal thread co-scheduling decisions and also to software that continuously optimizes the program itself. Nevertheless, in order to continually and automatically adapt systems resources to program behaviors and application needs, specific run-time information must be collected to adequately enable dynamic code optimization and operating system scheduling. Generally, run-time optimization is limited by the time required to collect profiles, the time required to perform optimization, and the inherent benefits of any optimization or decisions. Initial techniques for effectively utilizing run-time information for dynamic optimization and informed thread scheduling in future multithreaded architectures are presented.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance Analysis and Design Aids; D.4.1 [**Operating Systems**]: Process Management

General Terms

Performance, Design

Keywords

Performance counters, profiling, scheduling, multithreading

1. INTRODUCTION

By leveraging the advances in semiconductor technologies, system developers are exploring new paradigms of System-on-a-Chip (SoC) processors, Chip Multiprocessors (CMP), and Multithreaded (MT) architectures. The evolution dictates that future high-performance systems will integrate tens of multithreaded processor cores on a single chip die, resulting in hundreds of concurrent program threads sharing system resources. These designs will be the cornerstone of not only high-performance computing and server environments, but will also emerge in general-purpose and embedded domains. Managing hundreds of threads requires continuous optimization of both system resource decisions and thread execution, for which hardware-only techniques are not sufficient. As such, it is critical to advance current systems software (operating system, run-time system, and compiler) technologies for these emerging machines. Although it is important for systems software to understand the complete view of multiple cores, it is first necessary to build effective models of multithreaded core execution that will likely be the basis for the multi-core designs.

Multithreaded architectures address the growing processor-memory gap by supporting multiple hardware thread contexts capable of hiding memory latencies of individual threads. Coarse Grained Multi-Threaded (CGMT) processors issue instructions from a single thread each cycle and switch between threads on long latency instructions such as cache misses or on definable time intervals. Alternative hardware thread contexts can perform useful work, increasing throughput, where a single thread would stall a processor. IBM released the PowerPC RS64-IV [6] which is a commercial implementation of a course grain multithreading processor. Simultaneous Multithreaded (SMT) [17][31][30] processors share the resources (ALUs, branch target buffers, etc.) of one physical processor between multiple "virtual processors" that simultaneously execute each cycle. The SMT design is intended to have a low design overhead for out-of-order processors, allowing it to be added into existing processor designs without significant cost. It was estimated that adding SMT support to the Compaq Alpha EV8 processor only required an additional 5% to the die area, and researchers at Intel found similar costs for their implementation of SMT called Hyper-Threading [19].

The most commonly available SMT processor is the Intel Pentium-4 processor with Hyper-Threading [13]. Hyper-Threading is technically similar to the SMT designs described in the research literature, although it has unique characteristics – in particular, certain physical resources are partitioned between the virtual processors while others are shared. Support for multithreading is enabled by the multiprocessor configuration tables in the ACPI (Application Configuration and Power Interface). When running a conventional operating system on a Pentium-4 Xeon system with Hyper-Threading enabled, each virtual processor appears to the operating system as two distinct processors and the base operating system does not need to have detailed knowledge that certain processors are in fact logical processors.

Despite efforts at enabling transparency in multithreaded processors, there exists significant potential in the operating system to be aware of the multithreaded model. More importantly, as multithreaded multi-core systems emerge, it will be increasingly important for operating systems to continuously monitor application behavior and assess job scheduling opportunities. To explore the space of this work, we evaluate SMT processors with the intent of providing the initial results and rationale of enabling operating systems for multithreading. Figure 1 shows a matrix of speedup values for different SPEC CPU 2000 applications run with reference input sets on a 2.53GHz Intel Pentium-4 using the "Northwood" processor design. The speedup is greater than 1 for applications pairs where the time to completion for the application is less when Hyper-Threading is enabled. When the speedup is less than 1, it is more efficient to run the applications sequentially or using a uniprocessor. Most application pairs either achieve little speedup or achieve some speedup – up to 30% speedup in some cases. However, there are some applications pairs that achieve significant slowdowns – as much as a 30% slowdown. In order to improve multithreading systems, performance-aware scheduling is required.

Like operating systems, run-time optimization systems for future processors can deploy optimizations, guided by profile information, to improve performance. However, in order to maximize the performance gain of these run-time optimizations, efficient profiling techniques are required that can accurately describe a program's runtime behavior. Profiling provides valuable information to a whole class of optimizations: superblock formation [12], code positioning [22], and improved function inlining[11]. The ideal run-time profile collection system has three distinct characteristics. First, it should provide accurate profile information for a dynamic optimizer to utilize. Second, the system ideally would gather all profile information in one stage. Finally, and most importantly, the run-time collection of information should occur with little to no overhead. Unfortunately, most approaches to profiling only meet one or two of these three goals. Instrumentation-based techniques provide an accurate profile while sacrificing the cost of overhead as well as convenience of compilation. Novel hardware-based techniques are emerging to collect run-time events using *hardware performance counters*. Although such structures efficiently capture run-time information, researchers have only begun to study the characteristics and benefit of the amount and type of information needed for driving run-time optimization [7, 18].

Modern microprocessors such as the Intel Pentium-4, Intel Itanium, and IBM PowerPC 970 provide a rich set of performance counters. Hardware performance monitoring units are commonly placed onto microprocessors to provide software engineers with low overhead means of performance tuning. These PMUs are usually fairly simplistic allowing for PC sampling and counters for certain events. The Intel Pentium-4 [29] provides a set of 18 event counters which can collect 50 different events. The Apple Computer Hardware Understanding Development(CHUD) Tool [2] can be used to sample the G5 performance counters.

This paper illustrates the potential of using low-overhead HPM (Hardware Performance Monitoring) information in scheduling and optimization in multithreaded processor cores. It is demonstrated that using the run-time information requires substantial analysis to construct effective algorithms and heuristics that aid systems software. The rest of this paper is organized as follows. Section 2 discusses related work in multithreaded scheduling and run-time profiling. Section 3 presents the potential of using modern hardware-based profiling techniques in run-time optimization. Section 4 gives an overview of constructing an accurate SMT-resource model using hardware counter information and in turn the implementation of that model in a performance-guided multi-threaded job scheduler. Section 5 describes the effectiveness of the scheduler, and conclusions are presented in Section 6.

2. RELATED WORK

2.1 Multithreaded Scheduling

Operating systems have a direct role in the performance of multithreaded machines when the number of software threads exceeds the number of hardware thread contexts. Since contention on shared resources can cause variations in an multithreaded system, the throughput of machines benefit from job scheduling, which is the process of selecting among a set of application threads to simultaneously execute and share the processor resources.

The work on *thread symbiosis* by Snavely *et al* [26, 27, 28] is the closest to the idea presented in this paper. Snavely *et al* proposes an operating system mechanism for discouraging threads with poor performance pairings from executing with one another. The SOS (Sample, Optimize, Symbios) scheduler [27] performs as the name suggests. A set of processes are *sampled*, collecting information from performance counters. Following this, an optimized schedule is calculated based on performance counter attributes recorded during the sampling phase; a number of "pairing functions" are proposed. This yields a period of *symbiosis scheduling*, where jobs deemed to benefit from co-scheduling or "*Symbios*" are executed concurrently. Snavely proposes a set of heuristics based on intuition or knowledge of the system microarchitecture and show that certain heuristics (*e.g.* using data cache miss rate or IPC (instruction per cycle) to indicate

	ammp	applu	apsi	art	bzip2	crafty	eon	equake	fma3d	galgel	gap	gcc	gzip	lucas	mcf	mesa	mgrid	parser	perlbmk	sixtrack	swim	twolf	vortex	wupwise
ammp	1.17																							
applu	1.03	1.05																						
apsi	1.14	1.07	1.13																					
art	0.98	0.94	0.99	0.78																				
bzip2	0.97	1.02	1.00	0.86	1.18																			
crafty	1.04	1.13	1.07	1.01	1.11	1.06																		
eon	1.20	1.20	1.22	1.29	1.07	1.10	1.12																	
equake	0.99	1.01	1.02	0.86	0.94	1.16	1.15	1.04																
fma3d	1.09	1.09	1.10	1.07	1.02	1.08	1.15	1.05	1.08															
galgel	1.08	1.15	1.13	1.01	1.04	1.13	1.24	1.07	1.16	1.16														
gap	1.03	1.12	1.07	0.98	1.19	1.23	1.13	1.24	1.09	1.11	1.24													
gcc	0.92	0.95	0.94	0.79	0.99	1.01	1.02	0.94	0.98	0.95	1.05	0.86												
gzip	0.96	1.01	0.97	0.83	1.12	1.07	1.04	1.03	1.00	0.98	1.10	1.09	1.27											
lucas	0.98	0.97	1.01	0.86	1.01	1.13	1.10	1.01	1.03	1.06	1.14	0.90	0.98	0.92										
mcf	1.02	1.03	1.04	0.84	0.95	1.10	1.20	0.98	1.09	1.09	1.09	0.84	0.90	0.96	0.94									
mesa	1.13	1.21	1.17	1.17	1.09	1.12	1.15	1.17	1.15	1.23	1.17	1.03	1.05	1.13	1.22	1.20								
mgrid	1.01	1.00	1.04	0.89	0.99	1.07	1.00	0.96	1.06	1.06	1.08	0.95	1.01	0.92	0.94	1.16	0.92							
parser	1.08	1.10	1.11	0.98	1.02	1.09	1.17	1.06	1.11	1.10	1.12	0.96	1.01	1.03	1.03	1.22	1.03	1.11						
perlbmk	0.94	0.97	0.94	0.83	1.03	0.93	1.01	0.96	0.96	0.96	1.08	1.06	1.13	0.92	0.86	1.02	0.97	0.97	1.08					
sixtrack	1.19	1.27	1.23	1.29	1.10	1.18	1.26	1.17	1.23	1.27	1.19	1.02	1.06	1.17	1.26	1.28	1.21	1.29	1.01	1.29				
swim	0.97	0.92	1.01	0.76	0.94	1.04	1.23	0.90	1.04	1.00	1.05	0.90	0.94	0.88	0.84	1.25	0.86	0.98	0.92	1.31	0.73			
twolf	1.14	1.01	1.11	0.92	0.98	1.01	1.20	0.98	1.05	1.03	1.03	0.93	0.94	0.98	0.96	1.12	0.95	1.04	0.93	1.20	0.91	1.07		
vortex	0.97	1.00	0.99	0.86	1.10	1.07	1.05	1.04	1.00	1.01	1.15	1.00	1.10	0.99	0.95	1.06	0.97	1.00	1.03	1.09	0.94	0.94	1.05	
wupwise	1.05	1.12	1.07	1.00	1.07	1.17	1.15	1.11	1.09	1.18	1.18	0.99	1.05	1.03	1.15	1.18	1.03	1.13	0.90	1.19	1.06	1.05	1.06	1.02

□ >=1.30 ░ 1.29-1.25 ░ 1.24-1.20 ▒ 1.19-1.05 ▒ 1.04-0.95 ▓ 0.94-0.90 ▓ 0.89-0.80 ■ 0.79-0.70

Figure 1: Speedup comparison when running application pairs sequentially *vs.* concurrently on an Intel Pentium-4 processor with Hyper-Threading. The speedup is expressed as a percentage of of the concurrent execution over sequential execution. Shading has been added based on the range of speedup.

likely pairings) yield poor results. By comparison, this paper presents a methodical, statistical model that can be used to derive the scheduling "pairing function". More importantly, use of a one-time machine characterization avoids the need for any distinct "sampling" and "optimization" phases. Instead, sampling and characterization are continuous, which are essential given the limited speedups possible from such sampling mechanisms. Likewise, this paper illustrates actual operating system implementation and discusses building effective scheduling models for real processors.

There is a large history of scheduling mechanisms that try to exploit program characteristics to improve throughput – [27] has a good survey. These techniques exploit higher level characteristics of the process (such as communication or I/O accesses). Our work focuses on scheduling tasks based on observed execution behavior once they have been entered into in a queue – it simply adjusts the selection of jobs of equal priority. There has been limited work in co-scheduling system for SMT processors that is actually implemented in the operating system and evaluated on a commercial processor. The most similar study compares the SOS technique on the Tera MTA, and yields 10% speedups [26] on combinations of parallel programs using manual co-scheduling. The speedup rises from balancing parallel *vs.* serial sections of different multithreaded programs as well as fine-tuning machine resources. In the same way, the statistical model used in our system is related to work by Isci and Martonosi [15], which derives a linear combination of performance counters to validate an activity based power model. However, they did not consider the percentage of variation from specific counters nor did they consider interactions between counters.

2.2 Profiling with Hardware Counters

Specialized hardware profiling techniques have been proposed for collecting run-time profile information. Conte [8] examines using branch handling hardware coupled with the branch predictor to obtain branch information. Merten's [20] work explores using a branch behavior buffer for collecting branch profile data. These techniques incur a low overhead and can effectively gather data during one program run but suffer in accuracy because they are designed to collect edge profiles. The ADORE dynamic optimization system [7, 18] is one excellent example of directly using hardware information for dynamic trace generation. ADORE uses the Itanium-2 PMU (Performance Monitoring Unit) for collecting profile information aimed at improving data cache performance. The primary goal of the ADORE optimizer is to use the Itanium-2 PMU to detect a small amount of hot traces for optimization. While ADORE is interested in a few traces to optimize during run-time, future run-time systems will need to gather and exploit as much information possible from PMUs, correlating these samples and characterizing the nature of the PMU information with respect to program behavior.

Other sampling ideas originate from continuous profiling and optimization systems [1, 16]. These systems sample performance monitors for profile information to drive feedback-based optimization between application invocations. This paper demonstrates an important addition into the area of continuous program optimization by illustrating how future systems can make use of existing hardware monitoring units in run-time optimization.

3. PATH PROFILING USING HARDWARE MONITORING

Collecting run-time program information is critical to directing next generation processors. A number of optimization techniques can use run-time profile to adapt program behavior as well as the allocation of resources. However, hardware profiling has several issues which bring its effective use into question. Accuracy is critical to enabling effective optimization. More importantly, in order for profiling to be feasible in a run-time system, it must be done with minimal collection overhead. Finally, unlike software profiling, hardware profiling is not deterministic as it uses sampling of events which might occur at different time points in a program execution.

3.1 Performance Monitoring

The work in this section of the paper uses the Intel Itanium-2 PMU [14]. The Itanium-2 includes a set of counters which can be configured to count among 500 events. It also allows for sampling of Event Address Registers to capture recent data or instruction cache or TLB misses. In the following section, it is illustrated how the Itanium-2 PMU can sample the processor's branch execution to obtain accurate partial paths.

3.2 A Study of Using Hardware to Generate Path Profiles

Path profiling [3] has been shown to be an important form of profiling [4]. Path profiles correlate branches by keeping track of path execution counts instead of simple branch counts. However, path profiling usually comes with a significant increase in overhead(31% for path profiling versus 16% for edge profiling [3]). As such, using hardware to help generate path profiles has substantial promise. However, since hardware-collected execution information is limited in size and type, any collected information must be assembled and transformed to be put in a more usable form. Figure 2 illustrates the problem of adapting hardware monitoring information to the problem of run-time path profiling. The figure illustrates a hot path of a program's execution and the partial information (PMU trace) that is collected from performance monitoring units. Essentially the sampled regions of code do not indicate the complete path profile of the code region.

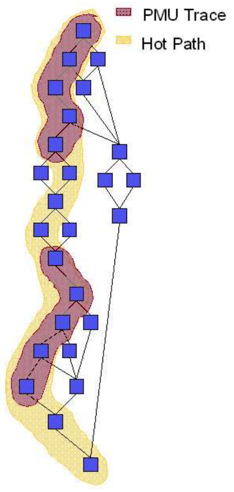

Figure 2: Path detection using PMU information.

At the center of the problem of hardware monitoring is that hardware has limited capacity to maintain all program information. For instance, the Itanium-2 PMU contains eight registers for collecting branch execution outcomes and the registers are treated as a circular buffer. Each executed branch instruction usually requires two of the BTB registers; one for the branch instruction address and another for the branch target address. Because of this, the BTB registers effectively act as a four branch circular buffer. In the Itanium-2 PMU, the user is able to conduct BTB samples through a set of user-defined filters. The following set of experiments use the SPEC 2000 benchmarks compiled with the base configuration of the OpenIMPACT Research Compiler [21]. A PMU collection tool based on the perfmon kernel interface and libpfm library [10] was constructed to collect samples of the Itanium-2 PMU taken-branch registers. The PMU samples are analyzed within an OpenIMPACT module to expand to larger intra-procedure paths.

3.2.1 *Effect of PMU Sampling Period*

Figure 3 shows the effect of sampling rate on run-time overhead as well as the number of unique paths discovered by the PMU for a few benchmarks. The sampling period is varied from 50K to 10M clock cycles. Naturally, a lower sampling rate decreases the overhead but provides a lower number of unique paths, while a high sampling rate increases the overhead but provides more unique paths. PMU sampling overhead remains relatively low, less than 10%, from 10M all the way down to around 500K. When the sampling rate is increased further, the percentage overhead increases quickly up to 50% for a sampling period of 50K. The number of unique paths discovered by the PMU rises steadily for each increase in sampling rate.

3.2.2 *Profile Determinism*

The profiling infrastructure enables aggregate profile information to be collected from multiple runs of a program and compared. By gathering PMU information over separate runs, analysis of lost paths due to statistical sampling can be measured. Figure 4 shows the effects of aggregating the PMU branch samples from multiple runs of a few benchmarks with the same input. The figure illustrates the additional runs increase the number of unique PMU paths. The greatest increase occurring from combining up to 10 runs, after which there is a slight leveling off. It is possible that the paths collected from multiple runs will fill in important partial paths that are missing from other runs. However, it is more important to understand if the paths collected accurately find the most important paths of program execution.

3.2.3 *Accuracy Results*

To measure accuracy of the PMU-generated paths, a full path profile was generated with a Pin tool. Pin was designed to provide functionality similar to the popular ATOM toolkit [9] for Compaq's Tru64 Unix on Alpha. Unlike ATOM, Pin does not instrument an executable statically by rewriting it before execution, but rather adds the code dynamically while the executable is running. This makes it possible to attach Pin to an already running process to collect profile information on the fly. However, Pin-instrumented binaries experience average slowdowns on the order of 1000% when collecting detailed information, and by themselves do not meet the profiling constraint of low overhead. The PMU path profile is compared to a full path profile gathered with

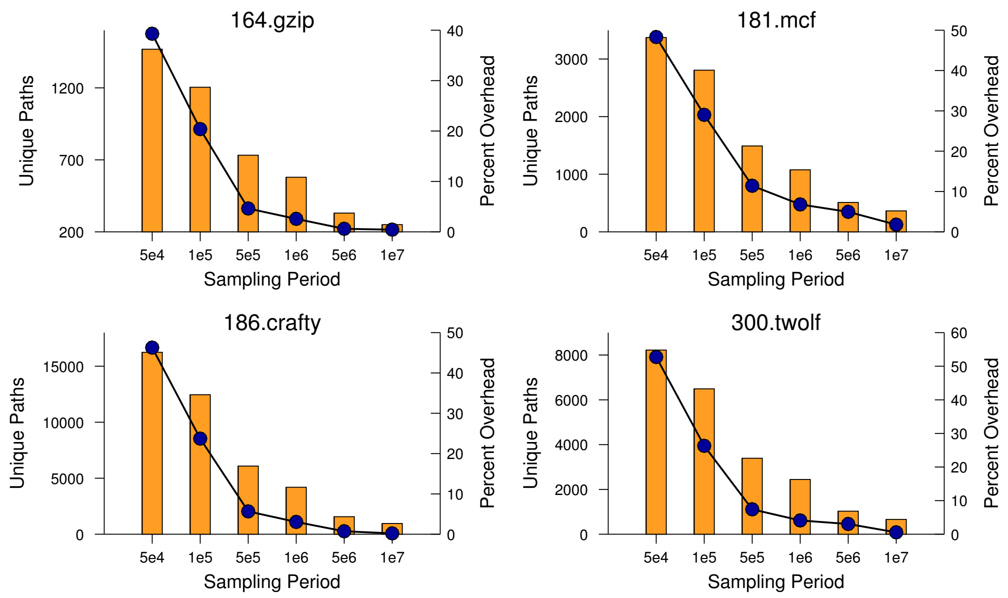

Figure 3: Overhead and number of unique paths for various sampling periods.

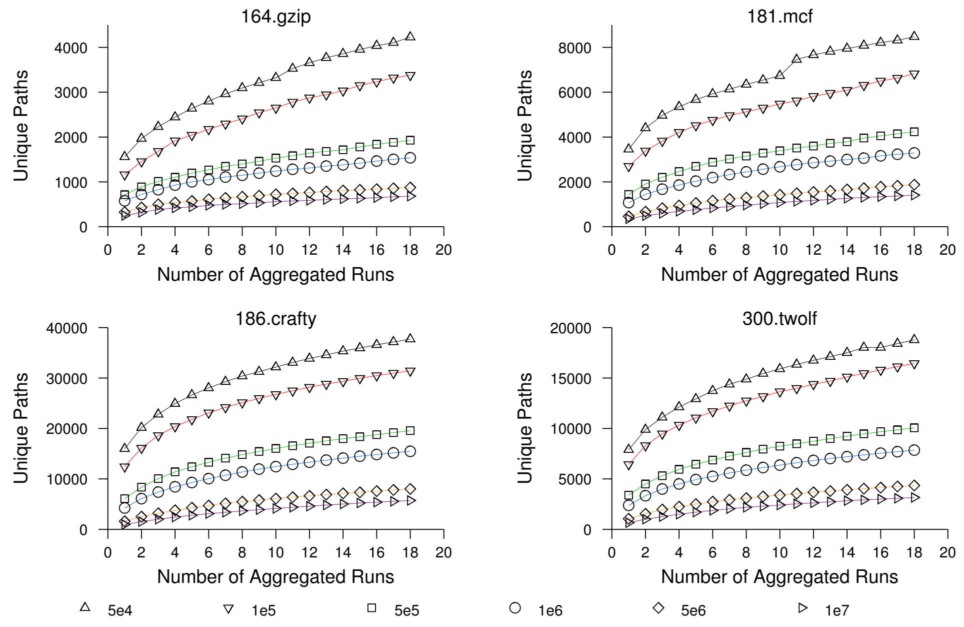

Figure 4: Number of unique paths found by aggregating data from runs with same input set.

a Pin tool using a method similar to Wall's weight matching scheme [32]. The accuracy is described as the fraction of estimated hot path flows as compared to hot path flows in the full path profile:

$$Accuracy\ of\ P_{estimated} = \frac{\sum_{p \in (H_{estimated} \cap H_{actual})} F(p)}{\sum_{p \in H_{actual}} F(p)}$$

In this equation $F(p)$ is the flow of a path. This is defined as the paths count divided by the count of all the paths added together. This represents the percentage of the all counts that path p accounts for. H_{actual} is the set of paths in the full path profile which are above a set threshold. A threshold of 0.125% is used, similar to previous path profiling studies [4, 5]. $H_{estimated}$ is then the created by selecting the hottest paths in our path profile equal to the number of paths in H_{actual}.

Figure 5 shows accuracy results using the method described above. In general, our accuracy ranges from 75% to 95%, averaging 80% accuracy. Applications *177.mesa* and *197.parser* are particularly bad with accuracies of 60%. Evidence shows that although our method performs well for some benchmarks such as *175.vpr*, *178.art*, and *183.equake*, it could use some improvement in others. Noise from the path matching scheme is most likely distorting the final path counts. Nevertheless, the data clearly motivates using hardware collected profile information for optimization methods that require even the most specific of information. The following section more closely examines the use of hardware information to impact multithreaded scheduling.

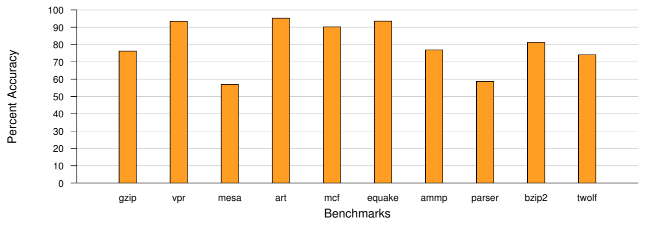

Figure 5: Accuracy of PMU-based path profiling.

4. SCHEDULING FOR SIMULTANEOUS MULTITHREADED ARCHITECTURES

The performance impact of operating system scheduling on multithreaded architectures depends directly on having an integrated run-time model of processor resources and application characteristics. For example, it is important for the scheduler to view logical processors in a multithreaded physical processor core as a dynamically varying pair of asymmetric processors. In the case that one logical processor is running a memory intensive application the other logical processor should be treated like a machine with fewer memory resources. Since the behavior of the processors depends on dynamically varying application demands and the current allocation of threads to logical processors, the scheduler determines the "properties" of a logical processor by profiling application demand and adjusting its internal scheduling model based on this profile. Building an effective run-time model for scheduling is the topic of the following section.

Consider a case where there are two logical processors (in a single 2-way multithreaded physical processor) and a total of R processes ready to execute. Furthermore, assume that it's possible to classify processes as either "good" (causing no or minimal slowdown when scheduled with most other processes) or "bad" (causing slowdown when scheduled with most other processes) and that there are $R = G + B$ of each of those processes available to run. At any scheduling interval, the probability of scheduling two "bad" jobs to the same physical processor concurrently using a random scheduler would be $(B * (B-1))/\binom{R}{2}$, and the probability of scheduling exactly one "bad" job would be $1 - \binom{G}{2}/\binom{R}{2} - (B * (B-1))/\binom{R}{2}$.

Since an operating system scheduler *must* eventually run all processes, the best outcome is to not run two "bad" jobs at the same time (this assumes that the scheduler always uses both logical processors). If running the full combination of the SPEC benchmarks in Figure 1 concurrently, then $G = 21$ and $B = 3$ (considering *176.gcc*, *171.swim* and *179.art* to be "bad" processes). This would result in about 2% of scheduling intervals where two "bad" jobs are scheduled concurrently, and about 22% of scheduling intervals where one "bad" program is run with a "good" program. If, as Figure 1 indicates, running a combination of "bad" jobs impacts performance by 20%. By detecting such "bad" jobs, the overall performance may be improved by 0.4% overall. The potential performance improvement increases when a smaller number of processes are involved. If $G = 2, B = 2$ then the overall performance might be improved by $33\% * 20\%$, or about 6%.

These potential speedups are small, but this simple analysis mainly indicates that any improved scheduling mechanism has to be efficient to be worth implementing. Microprocessor designers are hard-pressed to improve performance by more than 3-5% by any single architectural improvement. If a simple scheduling mechanism can achieve comparable gains, there is value in implementing it, as long as it does not *hurt* performance by co-scheduling "bad" processes more frequently than a random scheduler would.

Due to the need for extreme efficiency, this paper explores using *hardware performance counters* to predict how co-scheduled processes might interact. Most processors support performance counters for both debugging processor designs and gathering data for performance tuning applications. The organization in the Pentium-4 is similar to that of many other processors - the processor has a limited number of performance registers and a larger number of performance counters. A given performance counter can be associated with specific performance registers. Additional performance information can be synthesized from combinations of performance counters (for example, the instructions per cycle delivered by a processor can be calculated once the number of instructions retired and the processor cycles are counted). It is important to characterize the interaction of logical processors in an SMT system using a data-driven, empirical approach to determine which threads should be co-scheduled rather than adopt the *ad hoc* approach used in prior studies [27].

Five performance metrics are identified that indicate either particular aspects of program activity (such as the number of branches or floating point instructions) or implementation specific characteristics of the processor, such as branch mispredictions or trace cache lookup misses that occur due to incorrect processor speculation. These are shown in Table 1. These five metrics are chosen because they

b	Retired branches
tcm	Trace cache lookup misses
l2m	Second-level cache misses
f	Retired Floating point μops
ipc	Instructions per cycle

Table 1: Performance metrics recorded for application characterization. In reported metrics, events are normalized by the number of instructions issued to be per-cycle events counts. Thus, a value of 0.05 for `f` would indicate that 5% of cycles are spent on floating point instructions.

are a good representation of thread behavior given the constraints on performance counter allocation. Since there are two threads, each counter must be allocated twice, one per logical processor, and certain events can only be counted with certain counters. The Linux 2.6 task structure was modified to include software counters to shadow the hardware counters. Counter values are recorded during execution and can be used during scheduling.

Pair-wise combinations of the SPEC CPU 2000 benchmark suite ran with reference input sets and measured the performance counters of each application each time the scheduler was invoked. Using the R statistical computing package [23] to fit a linear model on this data and predict the IPC based on the sum of performance counters extracted from each application for each application pair. The motivation in using the *sum* of the performance counters was the intuition that most architectural mechanisms have some form of *capacity limit*. For example, the processor memory bandwidth is finite; if two processes tend to approach that limit, they will probably interfere with one another.

The goal of this was to determine if there is a simple set of performance registers that can be used to predict speedup (or slowdown); scheduling decisions could then use those specific set of performance counters and the derived model to predict which application pairings are most likely to yield speedups. It was separately observed that application behavior is reasonably consistent across scheduling quanta – in other words, that the immediate past is a reasonable predictor for the immediate future. Most programs exhibit some degree of *phase behavior* in the use of microarchitectural features [25]; this is one reason why the SOS technique of Snavely *et al* undergoes periodic resampling to determine which processes cooperate. The *autocorrelation coefficient* for the *ipc* and *l2m* counters were computed as 0.93 and 0.92 (*respectively*) for a single lag period, indicating that using the prior sample provides reasonable accuracy. Rather than use a larger sampling period and periodically re-sampled, the auto-correlation decreases, indicating less predictive accuracy – for example, summed over six scheduling quanta, the autocorrelation for *l2m* drops to 0.52. This implies that the resampling mechanism of Snavely *et al* would have higher overhead than our simpler mechanism.

The predictor for speedup was a linear model of each of the normalized performance counters that included multiplicative terms to capture interactions between microarchitectural features. The model is of the form

$$\begin{aligned} speedup &= w_0 + w_1 b + w_2 tcm + \ldots \\ &\quad + w_{10} cpc + w_{11} b * tcm + w_{12} b * l2m + \ldots \\ &\quad + residual. \end{aligned}$$

The full linear model includes a total of 256 terms. The weights for individual terms (w_i) define the contribution of a particular factor (e.g. b or tcm) or a combination of factors (*e.g.* $tcm{*}l2m$). The combinations of factors include interactions *between* specific factors – for example, $tcm * l2m$ represents the contribution of the *interaction* of trace cache misses and level-2 cache misses. This linear model has a multiple correlation coefficient of $R^2 = 0.942$, indicating that the model has very high predictive accuracy. The "residual" term represents any error term needed to have the linear model fit the data. The correlation coefficient can occasionally provide a misleading measure of the model accuracy. Further analysis was performed to verify that the statistical model was accurate.

The linear model serves three purposes. First, it indicates that it is possible to accurately predict speedup using samples from performance counter sampling of independent applications. Second, using the linear model, it's possible to determine *which* performance counters are most significant in predicting speedup. This can be done using two techniques. The first involves adding or removing terms from the linear model to see if a reduced set of performance counters yields a model with the same accuracy (defined by the R^2 metric); a complete analysis was performed by dropping individual performance counters and found that omitting any one counter reduced R^2 to values of $0.70 \ldots 0.30$. Thus, it appeared important to include this full set of performance counters.

The basic performance counters explain about 90% of the variation in the speedup when applications are paired. Based on the linear model, it should be possible to select applications by comparing a scaled sum of performance counters corresponding to the model – $-23 * l2m + 1.5 * f + 5.2 * b + -1 * tcm$. This simple heuristic captures the relative contribution of the leading contributors to the performance variation and involved only simple calculation, yielding an efficient solution.

4.1 Informed-Multithreaded Scheduling

Informed scheduling decisions are made in two ways: first by migrating tasks between processors so that any decision is more likely to be positive, and second by selecting tasks predicted to behave well with the other tasks currently running on the other logical processor. Previous work [24] illustrated that informing operating system schedulers with the information of processor performance counters has the potential of improving multithreaded architectures. This section examines the complete implementation of constructing an informed performance-guided multithreaded scheduler.

The Pentium-4 performance counters are configured prior to execution to record five separate metrics for each logical processor: branches, DTLB misses, L2 misses, floating point micro-ops, and instructions. Due to the constraints on performance counter allocation, it's not possible to directly record all the performance counters used in the linear model concurrently, and thus use this reduced set of counters. Each time the scheduler is invoked, these counters, along with the time stamp (in nanoseconds) are read and set to zero. After reading, the value is multiplied by a large integer factor before being divided by the elapsed time so that a scaled metric per cycle is created without using the floating point unit within the kernel. The adjusted value is then stored with the task structure of the previous task, providing an estimate of the events per cycle for the previous scheduling quantum. New processes have null values for these counters, meaning they will be scheduled ahead of most other processes; this is corrected once that process executes and records the performance counter values.

Our scheduler modification is done by adding a hook into the default Linux 2.6 scheduler that calls a function if it is registered by a module. The module handles all instrumentation, task selection, and the decision of when to migrate tasks, but the migration is part of the statically loaded kernel because the scheduling data structures are not exported. The Linux 2.6 scheduler uses a queue based scheme, and the first task in the queue has the highest priority to be scheduled next. In order to maintain scheduling fairness and responsiveness, but still exploit differences in processes, the scheduler only looks at the first four tasks on the run-queue and never skip a task more than three times.

The linear model is useful for predicting IPC and giving us information about what resources are most critical for scheduling. However, it is less useful for predicting actual speedup of paired tasks because of the residual effect between the actual counters and IPC. For example, the model predicts that a high floating point count will yield a high IPC, but it is obvious that pairing two applications with high floating point is the wrong decision. Instead of directly applying the model, its factors are used as a heuristic to guide scheduling. Instead of summing the counters from each process, the absolute difference is taken and weighed to indicate the difference. This scheme will pair jobs with different usage patterns. The following prediction function was derived. Assume process A is running on logical CPU 0, and a decision between other processes for co-scheduling is required. Each process has a set of counters representing the event count per cycle – for example B_r, B_{l2m} and so on. For each process, value is calculated: $v = 4(abs(A_{l2m} - B_{l2m})) + (abs(A_{tcm} - B_{tcm})) + (abs(A_f - B_f)) + 2(abs(A_b - B_b))$. The process among the first three in the priority queue with the highest v is selected. Notice that the scaling for the *l2m* term is four-fold that of the b and f terms – this is done because the model indicates that the *l2m* contributes 20% of the variance compared to the other counters. A similar argument holds for the *ipc* counters, but those counters are subtracted from the others since IPC is the only “higher is better” counter.

In a situation where jobs are randomly distributed between run queues, it is possible to encounter a situation where there is little choice but to schedule two processes that are cache-intensive. Our solution to this was to add a two bit saturating counter to each task structure. The counter is incremented after a “high-cache” interval, and decremented after a “low-cache” interval. When a task's two bit counter is maximized, it is migrated to a CPU that has been designated as the ”high-cache” CPU, if it is not already located there, and then the affinity mask is set so that process cannot be migrated away. When tasks on the ”high-cache” CPU fall to a weak state in the two bit counter the affinity mask is restored and the Linux scheduler can migrate it to another processor if necessary for load balancing. Fortunately, the Linux scheduler does an adequate job of load balancing, and as such no additional compensation is performed to account for tasks the algorithm removes. In practice, this method was very effective in grouping tasks; on average there were two or three tasks of a set of eight that were fixed at any given time. To prevent an imbalance, the number of fixed tasks was capped at half of the total runnable task count.

5. CO-SCHEDULER RESULTS

All measurements and experiments reported were run on a single 2.53GHz Pentium-4 “Northwood” workstation with 768MB of RDRAM using a modified Linux 2.6.5 kernel. All benchmarks were executed using a reference data set. Each experiment involved executing eight randomly selected benchmarks for a fixed time interval of four minutes on both the base Linux scheduler and our modified co-scheduler. Speedup was measured using the throughput IPC (measured in IA32 instructions per cycle rather than μops/cycle) of the processor.

The time limit was chosen for several reasons. First, it seems that the best way to evaluate this system would be to allow jobs to run to completion, but doing so makes it difficult to understand the results. In a set of jobs, suppose that all jobs except the slowest one improve; the system has obviously increased throughput, but the time to completion remained the same. Summing individual benchmark execution times is also flawed; improvement in shorter benchmarks has a cascading effect on the results of longer benchmarks. Also, since the timeslice in the Linux 2.6 kernel is 100ms for IA32 systems, the scheduler will make at least 2400 decisions in each experiment.

Figure 6 shows the speedup for fifty random sets of eight different applications drawn from the SPEC2000 benchmark suite. The speedup reported is an average of three measurements of four minute intervals of program execution. We ran the same combination of benchmarks using the default, unmodified scheduler in Linux 2.6.5 and our own modified version for the same period of time and then compare the throughput of the processor. For eight co-scheduled processes, we achieve an average speedup of 6.0%, with values ranging from -2.9% to 58%.

Figure 7 gives a closer look at the worst, median, and best benchmark sets. In these three examples, along with every other experiment, there is a severe disparity between the amount of speedup individual applications receive, even though the overall speedup is positive. It is important to note that although processes are chosen to run in a different order, they always receive the same amount of time on the processor. Since the scheduler is fair with respect to time, this implies that there is an issue of microarchitectural unfairness that must be addressed.

6. CONCLUSION

This work demonstrates that there is good potential to improve throughput using hardware-monitoring-based cooperative scheduling for multithreaded processors. Across fifty random sets of eight different applications, an average speedup of 6%, with some positive speedup values ranging from 11% to 25%. The results show a clear potential for operating systems to influence future systems, and more im-

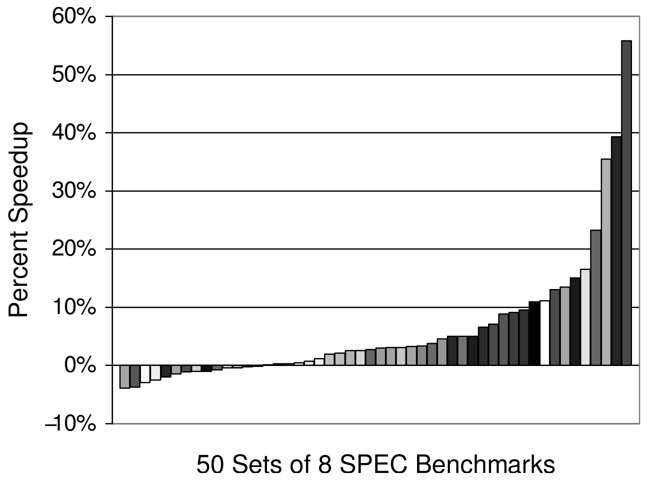

Figure 6: Speedup realized by each set of benchmarks over the base Linux scheduler.

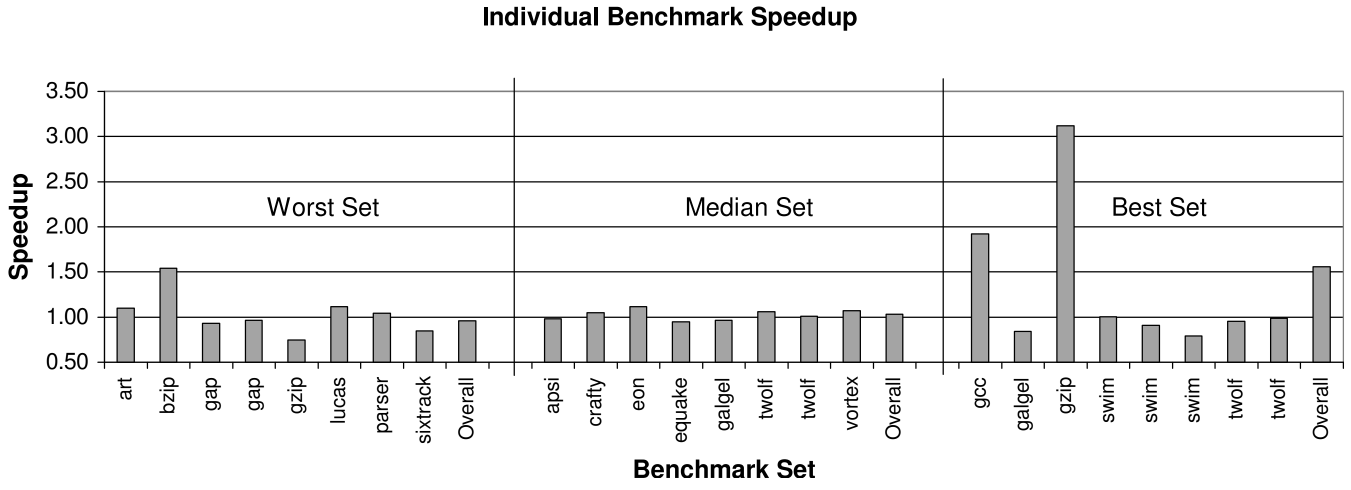

Figure 7: Individual benchmark speedups for sets with the worst, median, and best speedup.

portantly for performance counters to be integrated into operating systems. However, the improved throughput is distributed very unevenly; threads are sometimes penalized due to architectural unfairness, some receive dramatic improvement over the default scheduler and others only receive a few percent. A mechanism to evaluate how well a job "should" be doing is a novel problem in operating system scheduling. Traditionally, operating systems have scheduled "time," but increased sharing of resources may require scheduling for a combination of resources simultaneously. We believe the methodology used for selecting "pairing functions" can be applied to larger future systems with many SMT cores, or to systems with different microarchitures.

This paper also presented a system for extending hardware-collected information related to run-time optimization. The construction of a run-time hardware-based profiling system demonstrated that path profiling information can be accurately estimated (between 80-90%) while requiring very little performance overhead (between 3-5%). Overall the scheduling and profiling results illustrate the promising potential of performing continuous optimization of future processor resource decisions and thread execution.

7. REFERENCES

[1] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. of the 16th ACM Symposium of Operating Systems Principles*, pages 1–14, October 1997.

[2] Apple Computer, Inc. http://developer.apple.com/tools/performance/.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of 29th Annual Int'l Symposium on Microarchitecture*, pages 46–57, December 1996.

[4] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 134–148, January 1998.

[5] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizer. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization(CGO-2005)*, March 2005.

[6] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla,

and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, November 2000.

[7] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the International Symposium on Code Generation and Optimization(CGO 2003)*, March 2003.

[8] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24(2):187–206, April 1996.

[9] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.

[10] Hewlett-Packard Development Company. perfmon project http://www.hpl.hp.com/research/linux/perfmon/.

[11] W. W. Hwu and P. P. Chang. Inline function expansion for compiling realistic C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 246–257, June 1989.

[12] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[13] Intel Corporation. Special issue on intel hyperthreading in pentium-4 processors. *Intel Technology Journal*, 1(1), January 2002.

[14] Intel Corporation. Intel Itanium 2 processor reference manual: For software development and optimization. May 2004.

[15] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 93. IEEE Computer Society, 2003.

[16] T. Kistler and M. Franz. Continuous program optimization. In *IEEE Transactions on Computers vol. 50 n. 6*, June 2001.

[17] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[18] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *Journal of Instruction-Level Parallelism 6(2004)*, pages 1–24, April 2004.

[19] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, Feb. 2002.

[20] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, J. C. Gyllenhaal, and W. W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proc. 2000 Int'l Symp. on Computer Architecture*, pages 136–147, June 2000.

[21] OpenIMPACT Research Compiler. http://www.gelato.uiuc.edu/.

[22] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[23] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004. 3-900051-07-0.

[24] A. Settle, J. Kihm, , A. Janiszewski, and D. Connors. Performance analysis of simultaneous multithreading in a powerpc-based processor. In *Proceedings of the International Conference on Parallel Architectures and Compiler Techniques*, October 2004.

[25] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349. ACM Press, 2003.

[26] A. Snavely and L. Carter. Symbiotic jobscheduling on the tera mta. In *Workshop on Multi-Threaded Execution Architecture and Compilers*, Jan 2000.

[27] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 234–244. ACM Press, 2000.

[28] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 66–76. ACM Press, 2002.

[29] B. Sprunt. Pentium 4 performance-monitoring features. In *IEEE Micro 22(4)*, pages 72–82, 2002.

[30] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[31] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 54–58, 2000.

[32] D. W. Wall. Predicting program behavior using real and estimated profiles. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 59–70, June 1991.