Journal of Scheduling manuscript No.
(will be inserted by the editor)

A Computational Study of Heuristic and Exact Techniques for Superblock Instruction Scheduling

Michael Chase · Abid M. Malik · Tyrel Russell · R. Wayne Oldford · Peter van Beek

Received: date / Accepted: date

Abstract Compilers perform instruction scheduling to improve the performance of code on modern computer architectures. Superblocks—a straight-line sequence of code with a single entry point and multiple possible exit points—are a commonly used scheduling region within compilers. Superblock scheduling is NP-complete, and is done suboptimally in production compilers using a greedy algorithm coupled with a heuristic. Recently, exact schedulers have also been proposed. In this paper, we perform an extensive computational study of heuristic and exact techniques for scheduling superblocks. Our study extends previous work in using a more realistic architectural model, in not assuming perfect profile information, and in systematically investigating the case where profile information is not available. Our experimental results show that heuristics can be brittle and what looks promising under idealized (but unrealistic) conditions may not be robust in practice. As well, for the case where profile information is not available, some methods clearly dominate. Notably, a much inferior method is deployed in at least one existing compiler.

Keywords Superblock scheduling · Instruction scheduling · Compiler optimization · Profile-directed optimization

M. Chase
Cheriton School of Computer Science, University of Waterloo

A. M. Malik
Cheriton School of Computer Science, University of Waterloo

T. Russell
Cheriton School of Computer Science, University of Waterloo

R. W. Oldford
Department of Statistics & Actuarial Science, University of Waterloo

P. van Beek
Cheriton School of Computer Science, University of Waterloo,
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1
E-mail: vanbeek@cs.uwaterloo.ca

1 Introduction

Modern computer architectures have complex features that can only be fully taken advantage of if the compiler schedules the compiled code. This instruction scheduling, as it is called, is one of the most important steps for improving the performance of object code produced by a compiler as it can lead to significant speedups [18]. A standard region of code for scheduling in an optimizing compiler is called a superblock[1]. A superblock is a straight-line sequence of instructions with a single entry point and multiple possible exit points. A quite unique feature of superblock scheduling is that each of the exit points has associated with it a probability that the flow of control will exit the superblock at this point. In other words, with some probability, the remainder of the schedule of instructions will not be executed. The cost function of superblock scheduling includes these exit probabilities and the goal is to minimize the expected number of instructions executed.

Superblock instruction scheduling for realistic multiple-issue processors is NP-complete [21] and currently is done using the well-known list scheduling algorithm coupled with a priority heuristic in most, if not all, commercial and open-source research compilers. Recently, however, exact schedulers based on enumeration and on constraint programming have also been proposed [32,28]. Two classes of methods are used to determine the exit probabilities that appear in the cost function of superblock instruction scheduling: dynamic methods and static methods. In dynamic methods, the exit probabilities are estimated by executing the instructions on representative data, a process known as profiling. In static methods, the exit probabilities are labeled using a heuristic or some fixed policy. Dynamic methods can give better estimates of the exit probabilities, but it has been noted that

[1] See Section 2 for detailed definitions and explanations of terms in computer architecture and instruction scheduling.

application developers are often reluctant to use profiling because of the engineering effort involved, the often dramatic increase in compile time—a compilation that can take seconds or minutes when no profiling is performed can take hours when profiling is performed—and a general lack of trust in the process [15, p. 339].

In this paper, we perform an extensive computational study of heuristic and exact techniques for superblock instruction scheduling. A notable feature of our experimental study is that we use realistic architectural models for scheduling, whereas most previous empirical comparisons have assumed idealized architectural models[2]. In particular, we address the following three experimental questions regarding the performance of superblock scheduling techniques.

Q1. In the case where the exit probabilities are estimated dynamically, how sensitive are the superblock schedulers to the accuracy of the profile information?
The best possible case for superblock instruction scheduling is when there is perfect profile information—i.e., the exit probabilities used to schedule the superblock are identical to those that occur when the superblock is executed. However, perfect profile information is not a realistic scenario as such accuracy is not attainable in practice. Surprisingly then, all previous empirical comparisons of scheduling techniques for superblocks have assumed perfect profile information. In our experiments we develop a realistic error model for profile information and examine whether the scheduling techniques are robust in the presence of the inevitable profiling inaccuracies.

Q2. In the case where the exit probabilities are estimated statically, how sensitive are the superblock schedulers to the choice of fixed policy?
Previous work has performed empirical comparisons of only a few methods for statically estimating the exit probabilities and only for a few scheduling techniques. In our experiments we systematically vary the fixed policy for labeling the exit probabilities and examine whether there are policies that dominate others in terms of expected performance.

Q3. Are dynamic methods for estimating the exit probabilities worth the considerable extra effort or are static methods sufficient?
As noted above, application developers are often reluctant to use profiling. A question that has not been systematically addressed in previous work is whether profiling is worth the effort. In our experiments we compare the expected performance loss of the best fixed policy versus the case where profiling is performed.

[2] See Section 3 for a detailed comparison to previous experimental studies.

The three most important lessons learned from our study are the following. First, for the case where profile information is available, our experimental results show that heuristics can be brittle for architectures with small issue widths—a measure of how many instructions can be initiated in each clock cycle—and what looks promising under idealized (but unrealistic) conditions may not be robust in practice. An exact scheduler and one of the heuristic schedulers were found to be the most robust in the presence of (necessarily) inaccurate profile information. The exact scheduler was always better but is more costly in terms of scheduling time whereas the heuristic scheduler is fast and relatively accurate. Second, for the case where profile information is not available and exit probabilities are labeled using a fixed policy, some methods clearly dominate. Notably, one of the much inferior policies is deployed in at least one existing compiler. Finally, for architectures with smaller issue widths profiling can be important and lead to better schedules than the static methods we examined. However, for larger issue widths this is no longer true and profiling does not lead to significant performance improvements over static methods. Given that profiling can be difficult and time consuming, an important lesson from our experiments is that the faster and easier-to-use static methods may be preferred for larger issue architectures.

2 Background

In this section, we review the necessary background in computer architecture before defining the superblock instruction scheduling problem (for more background on these topics see, for example, [18,22,30]).

2.1 Computer architecture

We consider a standard multiple-issue, pipelined processor. In such a processor, there are multiple functional units and multiple instructions can be issued (begin execution) in each clock cycle. Examples of functional units include arithmetic-logic units (ALUs), floating-point units, memory or load/store units that perform address computations and accesses to the memory hierarchy, and branch units that execute branch and call instructions. The number of instructions that can be issued in each clock cycle is called the *issue width* of the processor. On most architectures, the issue width is less than the number of available functional units.

Pipelining is a standard hardware technique for overlapping the execution of instructions on a single functional unit. A helpful analogy is to a vehicle assembly line [22] where there are many steps to constructing the vehicle and each step operates in parallel with the other steps. An instruction is issued on a functional unit (begins execution on the

pipeline) and associated with each instruction is a delay or *latency* between when the instruction is issued and when the instruction has completed (exits the pipeline) and the result is available for other instructions that use the result. Also associated with each instruction is an *execution time*, the number of cycles between when the instruction is issued on a functional unit and when any subsequent instruction can be issued on the same functional unit. An architecture is said to be *fully pipelined* if every instruction has an execution time of 1. However, most architectures are not fully pipelined and so there may be cycles in which instructions cannot be issued on a particular functional unit, since the unit will still be executing a previously-issued instruction.

Further, many processors contain *serializing instructions*, instructions that require exclusive access to the processor in the cycle in which they are issued. This can happen when an architecture has only one of a particular resource, such as a condition register, and needs to ensure that only one instruction is accessing that resource at a time. In the cycle in which such instructions are issued, no other instruction can be executing or can be issued—for that one cycle, the instruction has sole access to the processor and its resources.

Example 1 Consider an Intel Itanium processor [27,22]. The processor has nine functional units—two ALUs, two floating-point units, two load/store units, and three branch units—and an issue width of six. On this processor a floating-point *addition* instruction is fully-pipelined and has an execution time of 1 cycle and a latency of 4 cycles. Thus, a floating-point addition instruction can be issued at each cycle on a floating-point unit, but no other instruction can use the result of that addition until 4 cycles have elapsed. In contrast, the floating-point *division* and *square root* instructions are not fully-pipelined (i.e., the execution time is greater than one cycle) and so execution of these instructions lock up a floating-point unit and prevent other instructions from being issued on the same floating point unit. As well, the Intel Itanium contains serializing instructions (see [27, p.2:15]).

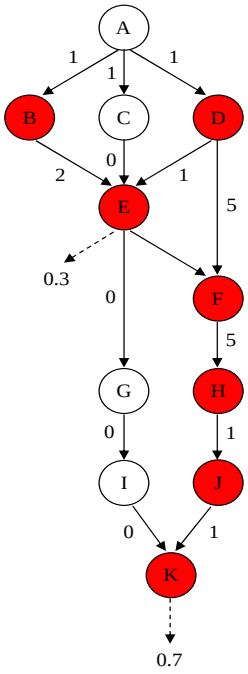
A compiler needs an accurate architectural model of the target processor that will execute the code in order to schedule the code in the best possible manner. In the rest of the paper, we refer to an architectural model as *idealized* if it assumes that (i) the issue width of the processor is equal to the number of functional units, (ii) the processor is fully pipelined, and (iii) the processor contains no serializing instructions. An architectural model is referred to as *realistic* if it does not make any of these assumptions.

2.2 Instruction scheduling

Instruction scheduling is done on certain regions of a program. All compilers schedule *basic blocks*, where a basic block is a straight-line sequence of code with a single entry point and a single exit point. However, basic blocks alone are considered insufficient for fully utilizing a processor's resources and most optimizing compilers also schedule a generalization of basic blocks called *superblocks*. A superblock is a collection of basic blocks with a unique entrance but multiple exit points [24]. We use the standard labeled directed acyclic graph (DAG) representation of a superblock. Each node corresponds to an instruction and there is an edge from i to j labeled with a non-negative integer $l(i,j)$ if j must not be issued until i has executed for $l(i,j)$ cycles. In particular, if $l(i,j) = 1$, j can be issued in the next cycle after i has been issued; and if $l(i,j) > 1$, there must be some intervening cycles between when i is issued and when j is subsequently issued. These cycles can possibly be filled by other instructions. Each node or instruction i has an associated execution time. *Exit nodes* are special nodes in a DAG representing the branch instructions. Let n be the number of exit nodes in a superblock. An exit i, $1 \leq i < n$, is called a *side exit* and exit n is called the *fall through exit*. Each exit node i has an associated weight or exit probability $w(i)$ that represents the probability that the flow of control will leave the superblock through this exit point. The probabilities can be estimated by running the instructions on representative data, a process known as *profiling*, or estimated statically using a heuristic or some fixed policy.

In profiling, an application is compiled a first time and executed on sample inputs. During this execution, the run-time behavior of the application, such as the probability that a branch is taken or how often an instruction is executed, is recorded. The application is then compiled a second time and the recorded run-time information is used to optimize the application. In the case of superblocks, the run-time information about the probability that a branch is taken and how often an instruction is executed are both used when forming the superblocks and the information about the probability that a branch is taken is used when scheduling the superblocks. It is also possible to form high-quality superblocks and schedule them without profiling information. Instead of determining the probability that a branch is taken using profiling, heuristics are used to predict the most likely direction of a branch [20]. Then, when scheduling the superblock, a fixed policy is used to assign the probabilities or weights that are associated with each exit in the superblock. The fixed policies have the form of assuming that every side exit is taken with percentage p, for some $0 \leq p < 100$, given that the flow of control has reached this branch. Given this percentage p, the $w(i)$ are readily constructed.

Example 2 Suppose $p = 50$ and consider a superblock with $n = 3$; i.e., two side exits $w(1)$ and $w(2)$ and a final fall through exit $w(3)$. The first side exit will be taken with probability $w(1) = 0.5$. The second side exit will be taken with probability $w(2) = 0.5 \times 0.5 = 0.25$; i.e., half the time the



(a)

cycle	S_1		S_2	
1	A		A	
2	C	B	C	D
3		D		B
4	G	E		
5	I		G	E
6			I	
7				F
8		F		
9				
10				
11				
12				H
13		H		J
14		J		K
15		K		

(b)

Fig. 1 (a) DAG representation of a superblock, where nodes E and K are exit nodes with exit probabilities 0.3 and 0.7 respectively, and (b) two possible schedules for Example 3.

flow of control reaches the second branch point and given that the flow of control has reached this branch point, there is a 50% chance that the flow of control will exit at this branch. Finally the fall through exit will be taken with probability $w(3) = 1 - 0.50 - 0.25 = 0.25$.

Given a labeled dependency DAG for a superblock and a target architectural model, a *schedule* for a superblock is an assignment of a clock cycle to each instruction such that the latency and resource constraints are satisfied. The resource constraints are satisfied if, at every time cycle, the resources needed by all the instructions issued or executing at that cycle do not exceed the limits of the processor.

Definition 1 (Superblock instruction scheduling) The *weighted completion time* or cost of a superblock schedule is $\sum_{i=1}^{n} w(i)e(i)$, where n is the number of exit nodes, $w(i)$ is the weight of exit i, and $e(i)$ is the clock cycle in which exit i will be issued in the schedule. The *superblock instruction scheduling problem* is to construct a schedule with minimum weighted completion time.

Example 3 Consider the superblock shown in Fig. 1. Nodes E and K are branch instructions, with exit probability 0.3 and 0.7, respectively. Consider an idealized processor with two functional units. One functional unit can execute shaded instructions and the other can execute unshaded instructions. Figure 1(b) shows two possible schedules, S_1 and S_2. The weighted completion time for schedule S_1 is $0.3 \times 4 + 0.7 \times 15 = 11.7$ cycles and for schedule S_2 is $0.3 \times 5 + 0.7 \times 14 = 11.3$ cycles. Schedule S_2 is an *optimal* solution.

In most, if not all, commercial and open-source research compilers, finding a schedule for a superblock is done using the well-known list scheduling algorithm. The list scheduling algorithm builds up a schedule cycle by cycle, maintaining a queue of instructions that are ready to be scheduled at a time cycle. The best instruction to schedule next is chosen using a priority heuristic. The heuristics generally consist of a set of features and an order for testing the features or a method for combining the features into a single value. Two example features are the *critical-path distance* from a node i to a node j in a DAG, which is the maximum sum of the latencies along any path from i to j, and the *earliest start time* of a node i, which is a lower bound on the earliest cycle in which the instruction i can be scheduled. Many hand-crafted heuristics have been proposed including dependence height and speculative yield (DHASY) [14,5], G* [7], speculative hedge [11], and balance scheduling [13]. As well, machine learning using decision trees has been applied to semi-automatically construct a priority heuristic [31]. As one example, the dependence height and speculative yield heuristic (DHASY) [14,5] weights the critical-path distances to each branch while accounting for the maximum delay in the graph. The priority of an instruction i is calculated as,

$$priority(i) = \sum_{b \in B(i)} w(b)(cp(1,n) + 1 - ((cp(1,b) - cp(i,b))$$

where $B(i)$ is the set of exit nodes that are descendants of i, $w(b)$ is the exit probability of branch b, $cp(1,n)$ is the critical-path distance between the root node and the (unique) fall through exit node, $cp(1,b)$ is the critical-path distance between the root node and exit node b, $cp(i,b)$ is the critical-path distance between instruction i and exit node b, and the difference $cp(1,b) - cp(i,b)$ is the latest cycle that instruction i can be issued without delaying the branch b. As a second example, the G* heuristic [7] uses a profile independent scheduler and a ranking method to schedule superblocks. In this heuristic, a superblock is scheduled using the critical-path heuristic. The rank for each exit point is then calculated by dividing the cycle in which the exit point is scheduled by the sum of the exit probabilities for the exit point under consideration and its preceding exit nodes. The exit nodes are sorted in ascending order. The final schedule for the superblock is obtained by taking an exit point from the sorted list one by one and scheduling it as early as possible with its predecessors.

As well, exact schedulers that find the optimal schedule (under certain restrictive assumptions) have also been proposed. Shobaki and Wilken [32] present an exact scheduler based on enumeration and pruning, while Malik et al. [28] present an exact scheduler based on constraint programming.

Table 1 Summary of previous experimental comparisons of scheduling superblocks using the list scheduling algorithm with a priority heuristic and using exact methods.

heuristic	*profiling*	*no profiling*[a]	*architecture*
DHASY [5]	perfect	—	idealized
G* [7]	perfect	—	idealized[b]
Speculative hedge [11]	perfect	$p = 0, 50$	idealized[c]
Balance scheduling [29]	perfect	$p = 1$	idealized
Decision tree [31]	perfect	$p = 1$	idealized[c]
exact	*profiling*	*no profiling*	*architecture*
Enumeration [32]	perfect[d]	—	idealized
Constraint prog. [28]	perfect	—	realistic

[a] The values represent the percentage of time that a side exit is taken, given that the flow of control has reached this branch (see Section 2.2).
[b] Idealized architecture but further assumes that the execution time and latency are equal for each instruction.
[c] Idealized architecture but does not assume that the issue width is equal to the number of functional units.
[d] Strictly speaking, the exit probabilities were not determined here by profiling, as the experimental setup used the gcc compiler's "guess-branch-probability" flag, which uses a randomized model to guess branch probabilities. But the effect is the same as the exit probabilities used in scheduling were the same as the probabilities used in evaluating the schedule.

3 Related Work

In this section, we review previous experimental studies of scheduling techniques for superblocks and of profile-directed optimizations beyond superblocks.

3.1 Scheduling superblocks

Although previous work has generally introduced a new list scheduling heuristic or exact method and then performed an experimental comparison to earlier proposals, our focus here is on just the experimental methodology used in the work.

Table 1 summarizes previous experimental comparisons in the literature, where we give the heuristic or exact method introduced in the cited work, and then summarize the experimental methodology along three dimensions: whether the experiments using profiling assume perfect profile information, what experiments were performed for the case where no profiling information is available, and whether the architectural model used was idealized or realistic. It can be seen that all previous empirical comparisons of scheduling techniques for superblocks assume perfect profile information; i.e., the exit probabilities used in scheduling the superblock are exactly the same as when evaluating the resulting schedule. The assumption of perfect profile information is unrealistic as it assumes that either: (i) the input(s) used when profiling and compiling the software (done by the developer of the software) are the same input(s) used by the eventual user of the program, or (ii) different inputs are used but they lead to the same profile information. Neither of these holds. As well, there has been little work on comparing fixed policies in the case where no profiling information is available. Previous work has considered $p = 0, 1, 50$ or schemes that are quite similar. For example, in the case of no profiling, Trimaran [6] sets $w(i) = \varepsilon^i$, $1 \leq i < n$, and $w(n) = 100$, where ε is a small, non-zero value. This is effectively the same as $p = 1$. However, there has been no systematic experimental comparison of fixed policies and the only study we are aware of examined $p = 0$ versus $p = 50$ on a small number of benchmarks. As well, almost all previous experimental comparisons assume an idealized architectural model. Because the comparisons are performed on idealized architectures, a heuristic may appear to perform better than it actually would in practice on real architectures, which have complex features. In particular, there is a risk that a heuristic may appear to be near-optimal when tested on a simplified architecture, but not at all near-optimal when actually executed on a real architecture. Our experiments on more realistic architectures are designed to test how well the heuristics would perform in practice. Finally, Conte, Menezes, and Hirsch [9] examine how the accuracy of the resulting profiling information influences the formation and subsequent scheduling of superblocks. While not an experimental comparison of scheduling techniques—they use a fixed but unspecified heuristic in their experiments—their work is notable for not assuming perfect profile information.

3.2 Profile-directed optimizations beyond superblocks

Although all previous experimental comparisons of scheduling techniques for superblocks have made the assumption of perfect profile information, profile-directed superblock schedule is just one instance of a general class of profile-directed compiler optimization (see, e.g., [19] and references therein). Here we review that broader set of work as it relates to the assumption of perfect profile information.

Wall [33], in one of the first such studies in the general area of profile-directed optimizations, examines the question of how well a profile from one run on a particular input predicts the behavior of an application when the application is run on different input. The answer is, "disappointingly badly". Wall [33] also shows experimentally that assuming a perfect profile can lead to inflated expectations of a profile-driven optimization. Wang and Rubin [34] show that for some interactive applications, profiles can vary considerably from one user or group to another. Wu [35] discusses how the profiling information used in optimization can be inaccurate for several reasons including that programmers may not update profile information after bug fixes and that early compiler optimization phases degrade the accuracy of exit probabilities for later optimizations (e.g., by changing control flow). Eeckhout, Vandierendonck, and De Bosschere [12], in their study of selecting representative inputs, show that different groups of inputs can lead to significantly different branch behavior. Hsu et al. [23] note that application developers often use reduced (smaller) input sets than would be used when the application is deployed in order to reduce profiling time and that for some applications, reduced input sets lead to significantly different profiles. Berube and Amaral [3,4] propose an evaluation methodology appropriate for profile-guided optimization based on cross-validation. They stress the points that (i) the choice of training and testing inputs can have a significant impact on measured performance, (ii) multiple inputs must be used during the training process, and (iii) the training sets must be different than the testing set to avoid inflating the performance results.

All of these studies strongly suggest that, in the context of profile-directed superblock scheduling, the exit probabilities that we assume when scheduling a superblock will often be quite different from the actual exit probabilities when the superblock is executed on many different inputs in practice. Thus, it is important to close the gap in the literature and compare the performance of the scheduling techniques for superblocks under this more practical setting.

4 Computational Study

In this section, we present our computational study. We begin by presenting the experimental setup that is common across the experiments.

Table 2 Target architectural models for scheduling. The issue width is the number of instructions that can be initiated each cycle. The other values are the number of functional units of that type. In the case of the single issue processor, the functional unit can execute all types of instructions; otherwise, a functional unit is restricted to executing instructions of the same type.

		functional units				
model	*issue width*	*simple integer*	*complex integer*	*load/ store*	*branch*	*floating point*
1-issue	1	1				
2-issue	2	1		1	1	1
4-issue	4	2	1	1	1	1
6-issue	6	2		2	3	2

In our study, we used the SPEC 2000 benchmark suite [http://www.spec.org] as test data. The SPEC 2000 suite is a collection of 26 diverse software applications and is widely used to evaluate new CPUs and compiler optimizations. The benchmarks were compiled with IBM's Tobey compiler and the superblocks were captured as they were passed to Tobey's instruction scheduler. The superblocks we used were obtained when instruction scheduling was performed after register allocation. As well, we only retained the superblocks which were executed at least 1000 times, as the more infrequently executed superblocks have little or no impact on the expected number of cycles executed by an application. This resulted in a total of 16,072 superblocks.

The superblocks contain four types of instructions: integer, load/store, branch, and floating point. The range of the latencies is: 1–37 for integer instructions (the largest value is for division), 1–12 for load/store instructions (the largest value is for a store-multiple instruction, which stores to memory the values in a sequence of registers), all 1 for branch instructions, and 1–38 for floating point instructions (the largest value is for square root). Scheduling was done for four realistic target architectures, summarized in Table 2.

The four target architectures are designed to be representative of existing architectures. Table 3 shows example processors and some of their current uses, for the various issue widths that we use in our experiments. Single and dual issue processors are widely used in embedded systems. For example, currently most smartphones use single issue processors, with more sophisticated smartphones now beginning to use dual-issue processors. As well, notebook and netbook computers often use dual-issue processors. Wider issue processors are used in higher end devices. For example, most desktop and laptop computers are 4-issue, with some enterprise and high performance computers using 6-issue processors. We note that the embedded market (i.e., smaller issue width processors) is much larger than the desktop and laptop market (i.e., wider issue width processors). For example, the ARM 11 processor design, a single issue processor, is perhaps the most widely used processor design across all computing devices. As of early 2008, 10 billion ARM 11

Table 3 Example processors, date of initial release, and example uses, for various issue widths, where the issue width is the number of instructions that can be initiated each cycle. All of these processors are currently still in production and are widely used.

width	*example processors*	*release date*	*example uses*
1	IBM PowerPC 405 [25]	June, 1998	Apple iPhone (original & 3G); Apple iPod (1 & 2); HTC, Nokia, and Samsung smartphones; Nintendo 3DS; Kindle DX; RIM Blackberry; GPS devices; smart meters; digital cameras; digital TV; printers, fax machines, network cards, media devices, storage devices, automobile controllers, ATMs
	ARM 11 [10]	April, 2002	
2	IBM PowerPC 460 [25]	October, 2006	Apple iPhone (3GS & 4); Apple iPod (3 & 4); Apple iPad (1 & 2); HP, HTC, Nokia, and Samsung smartphones; HP TouchPad; high-end storage and networking applications, digital signal processing (DSP), imaging, industrial control
	ARM Cortex-A8/A9 [2]	September, 2009	
	Intel Atom N570 [26, p. 2-17]	March, 2011	
4	Intel Core i7-2600 [26, p. 2-18]	January, 2011	Desktop & laptop computers
6	IBM PowerPC Power7 [25]	February, 2010	Enterprise servers & high performance computing
	Intel Itanium 9300 [27]	February, 2010	

processors had been sold and it is estimated that currently 5 billion are sold per year [1].

In contrast to most previous work, we do not assume an idealized architectural model. In our realistic architectural model the issue width of the processor is not equal to the number of functional units, the processor is not fully pipelined, and the processor contains serializing instructions. The instruction set, execution times, latency times, and serializing instructions are from the PowerPC architecture (see Table 3), a processor that is used in embedded systems and in servers. As an example of a non-pipelined instruction, the floating point square root instruction has an execution time of 32 cycles and a latency time of 32 cycles, which means that no other instructions can be issued on the floating point functional unit until the instruction has completed. As a second example, unsigned integer division has an execution time of 19 cycles and a latency of 20 cycles. Thus, after 19 cycles another instruction can be issued on the integer functional unit but the result of the division will not be available for other instructions to use until after 20 cycles. Approximately 15% of the instructions executed in our benchmark set of superblocks are serializing instructions.

We performed two sets of experiments: one set where the branch probabilities are estimated dynamically using profiling, and one set where the branch probabilities are set statically using a fixed policy. In both sets of experiments, we compare the various scheduling methods against a "gold standard" that is determined as follows. Each of the 26 benchmarks in the SPEC 2000 suite was compiled and then profiled using the training data set associated with that benchmark. The Tobey compiler then uses the profiling information to construct the exit percentage for each branch instruction and to determine the number of times each instruction is executed. It is these superblocks where the branches are labeled with exit probabilities and the instructions are labeled with frequency of execution that we capture and use in our experiments. For each of the 26 benchmarks, we optimally schedule each superblock in the benchmark using the constraint programming optimal scheduler [28] and determine how many cycles would be executed by the application when the scheduling is done optimally and the profile information is perfect. This is the gold standard (see, e.g., Table 5). No algorithm can do better than the gold standard and the question of interest is how far is a method from this gold standard?

In both sets of experiments, we evaluate a list scheduling algorithm using three heuristics: the dependence height and speculative yield (DHASY) heuristic [5], the G* heuristic [7], and the decision tree (DT) heuristic constructed using machine learning techniques [31]. The DHASY and G* heuristics are considered two of the best available superblock scheduling heuristics: these are the two heuristics that are made available in the Trimaran compiler [6] and Russell et al. [31] report that their experiments indicate these are the two best hand-crafted heuristics available for superblocks. We include the decision tree (DT) heuristic as Russell et al. [31] also report that this heuristic performs well in comparison to the DHASY and G* heuristics. We did not compare against the balance scheduling heuristic [29] because of its high computational cost and we did not compare against the speculative hedge heuristic [11] because of its quite poor performance in previous experimental studies [31].

In both sets of experiments, we also evaluate the exact scheduler based on constraint programming proposed by Malik et al. [28]. We did not compare against the exact scheduler proposed by Shobaki and Wilken [32] as it is applicable to idealized architectures only and it does not scale up to as large or difficult superblocks.

Our experiments were run on the SHARCNET Whale cluster, which consists of 768 machines running HP Linux XC 3.0, each with 4 GB of RAM and 4 2.2 GHz processors.

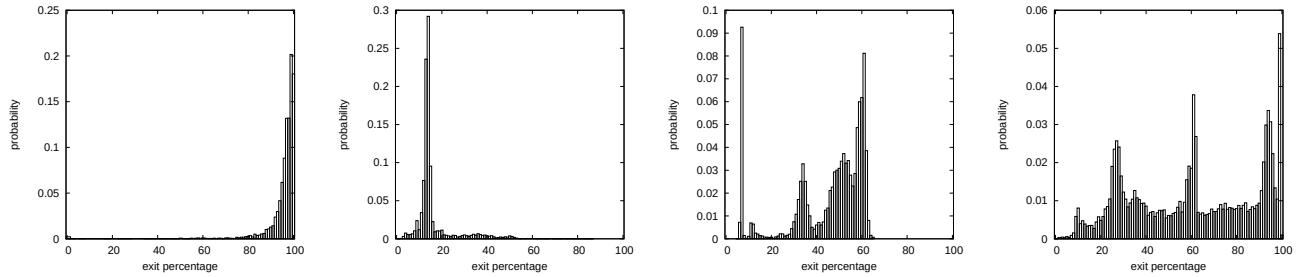


Fig. 2 Probability distributions for the exit percentages of four example branch instructions.

4.1 Experiment 1: Profiling

In our first set of experiments, we assume that the exit probabilities in the superblocks are determined using profiling. We study the effect of error in the estimates of the exit probabilities on the overall performance of the schedules.

Ideally, to systematically study the effect of the accuracy of the estimates of the exit probabilities, we would:

a. determine for many representative inputs or sets of inputs and for each superblock in our test suite, the probability each branch is taken if the superblock were executed on those inputs; and
b. schedule the superblocks based on the branch probabilities obtained and evaluate the resulting schedules against the gold standard: the optimal schedules based on perfect profile information.

Unfortunately, Step (a) of the ideal was not attainable in practice for three reasons. First, many of the applications in the SPEC 2000 benchmark suite come with only a single training input and additional inputs are not available. Second, our framework for obtaining superblocks and profile information did not easily allow us to determine the probability a branch was taken if the superblock were executed on arbitrary inputs. We used the Tobey compiler to obtain our testbed of superblocks. However, the Tobey compiler (as currently written) is constructed to use a given input for *forming* the superblocks as well as for obtaining the branch probabilities. In other words, each new profiling input leads to potentially different superblocks being constructed. This would confound our goal of comparing schedulers on fixed superblocks, where the superblocks scheduled are necessarily the same as the superblocks that are later executed by the users of an application when the application is invoked on their input. Finally, what constitutes a representative set of inputs must depend on actual usage and this could vary considerably between users of the same application.

Instead, the branch percentages in Step (a) are determined by sampling from empirical probability distributions. The empirical distributions were constructed using the gcov profiling tool and the gcc compiler applied to the MiBench benchmark suite [17], a collection of 32 application programs written in the 'C' language. Fortunately, for each of the 32 applications, 1000 inputs have been collected and are freely available [8]. We compiled and profiled each application on each of its inputs. This gave a collection of more than 8,000 branch instructions where for each branch we had a sample of up to 1000 percentages (sometimes the number of samples was less than 1000 as a branch was not always executed for each input, however the vast majority of the branches had at least 950 samples each). Interestingly, almost precisely a half of the distributions are point distributions; i.e., every one of the samples resulted in exactly the same percentage and thus there was no uncertainty associated with this branch. The remaining distributions ranged from distributions with small standard deviations to distributions with large standard deviations (see Figure 2).

Given that the empirical distributions have been constructed, suppose that in the perfect profile a side exit is taken $p\%$ of the time. Determining a percentage for this side exit in Step (a) proceeds in two steps. First, one of the 8,000 empirical distributions is selected at random, but proportionally to the probability mass associated with $p\%$ (i.e., if none of the samples had the branch taken $p\%$, the empirical distribution would never be chosen and if all of the samples had the branch taken $p\%$, the distribution would be much more likely to be chosen). Second, a percentage is selected at random from the chosen empirical distribution. The fall through exits are determined once the side exits have been randomly selected. The above process simulates profiling on a single representative input. Fisher and Freudenberger [16] show that profiling on a collection of inputs and taking the average of the exit probabilities to form a single hybrid profile can improve branch prediction accuracy. In our experiments, we simulate profiling on a set of representative inputs, by taking the average of 30 samples from the randomly selected empirical distribution. Our experimental methodology allows us to keep the superblocks fixed—as desired, as this is what would occur in practice—and to systematically sample from a representative set of branch probability distributions. Together, the large testbed of superblocks and empirical branch distributions should capture many of the different applications and inputs that arise in practice.

Table 4 Average and standard deviation of the percentage from gold standard on superblocks from each SPEC 2000 benchmark for the constraint programming scheduler (*copt*), the list scheduler with the decision tree heuristic (h_{dt}), the list scheduler with Bringmann's heuristic (h_{dhasy}), and the list scheduler with the G* heuristic (h_{g*}), for various issue widths, for the case where profiling is performed (without and with error in profiling information), and for the case where profiling is not performed (instead all side exits are labeled with exit percentage p). The entries for profiling with error represent profiling on a single randomly chosen input. The entries in grey indicate a percentage increase of 2% or greater.

		Profiling				*No profiling*											
width	*scheduler*	*no error*		*error*		$p=0$		$p=1$		$p=25$		$p=50$		$p=75$		$p=90$	
		ave.	*s.d.*	*ave.*	*s.d.*	*ave.*	*s.d.*	*ave.*	*s.d.*	*ave.*	*s.d.*	*ave.*	*s.d.*	*ave.*	*s.d.*	*ave.*	*s.d.*
1-issue	*copt*	0.0	0.0	0.2	1.1	8.8	22.2	5.2	15.6	0.7	1.4	0.8	1.3	0.9	1.4	1.0	1.4
	h_{dt}	0.4	0.7	1.8	7.9	10.0	25.3	9.8	25.3	4.8	15.7	3.8	14.6	2.5	7.4	2.6	7.3
	h_{dhasy}	1.2	1.1	2.5	8.2	11.6	25.6	10.6	25.7	7.0	18.3	4.5	14.7	2.8	7.4	3.1	7.6
	h_{g*}	0.7	0.8	1.7	6.0	11.6	25.6	11.6	25.6	3.3	7.3	2.4	7.4	2.8	7.4	2.9	7.4
2-issue	*copt*	0.0	0.0	0.2	0.7	5.6	20.3	2.9	9.8	0.4	0.6	0.4	0.6	0.5	0.6	0.5	0.5
	h_{dt}	0.8	0.9	1.1	2.1	7.0	23.5	6.6	21.8	1.5	1.4	1.1	1.1	1.2	1.1	1.1	1.0
	h_{dhasy}	1.0	1.0	1.4	3.0	7.3	21.5	6.7	21.6	1.8	1.6	1.3	1.1	1.3	1.1	1.3	1.1
	h_{g*}	1.1	1.4	1.5	4.7	7.3	21.5	7.3	21.5	1.5	1.2	1.2	1.3	1.3	1.2	1.3	1.1
4-issue	*copt*	0.0	0.0	0.1	0.2	3.0	10.3	0.6	1.1	0.2	0.2	0.2	0.3	0.3	0.4	0.3	0.4
	h_{dt}	0.7	0.9	0.9	2.0	3.4	10.2	3.4	10.2	1.0	1.1	0.9	1.1	1.0	1.1	0.9	1.0
	h_{dhasy}	0.8	0.9	1.0	2.0	3.6	10.3	3.4	10.4	1.2	1.1	1.0	1.0	0.9	1.0	1.0	1.0
	h_{g*}	0.7	0.9	1.0	2.7	3.6	10.3	3.6	10.3	1.1	1.1	0.9	1.2	1.0	1.1	0.9	1.0
6-issue	*copt*	0.0	0.0	0.0	0.1	3.1	11.3	0.4	0.6	0.3	0.4	0.2	0.3	0.2	0.3	0.4	0.8
	h_{dt}	0.7	1.0	0.8	2.2	3.4	11.2	3.3	11.2	1.0	1.3	0.8	1.1	0.8	1.2	0.8	1.1
	h_{dhasy}	0.8	1.1	0.9	2.2	3.6	11.2	3.3	11.2	1.0	1.3	0.9	1.1	0.8	1.1	0.9	1.1
	h_{g*}	0.6	0.9	0.8	2.8	3.6	11.2	3.6	11.2	1.0	1.1	0.8	1.1	0.8	1.2	0.8	1.2

Table 4 (columns under "Profiling") shows the average and the standard deviation of the percentage increase in the number of cycles executed compared to the gold standard on the superblocks from the SPEC 2000 benchmarks, for the various superblock schedulers and without and with profiling error. For the experiments with profiling error, we performed a minimum of 10 trials per benchmark application for the constraint programming scheduler (*copt*), and a minimum of 100 trials per benchmark application for the list scheduler with the heuristics h_{dt}, h_{dhasy}, and h_{g*}. The average percentage increase is the average over all 26 benchmarks (equally weighted).

Table 5 shows in detail the results for the 1-issue architecture and the cases where (i) there is no error in profile information; (ii) there is error as the branch probabilities are determined by simulating profiling on a single representative input; and (iii) there is error but the error is reduced as the branch probabilities are determined by simulating profiling on a set of 30 representative inputs. For example, for the case of the 1-issue architecture and no error, the average percentage increase for the dependence height and speculative yield heuristic h_{dhasy} is 1.2% (over all 26 benchmarks) and the maximum percentage increase is 3.9% (on the benchmark "eon", a probabilistic ray tracer). As a second example, for the case of the 1-issue architecture and error in profiling on a single input, the average percentage increase for the heuristic h_{dhasy} is 2.5% (over all 26 benchmarks) and the maximum percentage increase is 24.0% (on the benchmark "swim", a weather prediction program).

4.2 Experiment 2: No profiling

In our second set of experiments, we assume that the exit probabilities in the superblocks are set statically using a fixed policy. We study the effect of the method of labeling the exit probabilities on the overall performance of the schedules.

The fixed policies that we investigate all have the form of assuming that every side exit is taken with percentage p, for some $0 \leq p < 100$, given that the flow of control has reached this branch (see Example 2). Recall that previous work has considered $p = 0, 1, 50$ or schemes that are quite similar. An obvious extension is to consider additional values and in our experiments we tested the percentages $p = 0, 1, 25, 50, 75, 90$. Table 4 shows the average and the standard deviation of the percentage increase in the number of cycles executed compared to the gold standard on the superblocks from the SPEC 2000 benchmarks, for the various superblock schedulers and values of p. Once again, the average and standard deviation of the percentage increase is over all 26 benchmarks (equally weighted). Table 6 shows in detail the case of the 1-issue architecture and $p = 1, 50, 75$. For example, for the case of the 1-issue architecture and $p = 1$, the average percentage increase for the constraint programming scheduler *copt* is 5.2% (over all 26 benchmarks) and the maximum percentage increase is 81.0% (on the "swim" application). As a second example, for the case of the 1-issue architecture and $p = 50$, the average percentage increase for the *copt* scheduler is 0.8% (over all 26 benchmarks) and the maximum percentage increase is 6.3% (on the "sixtrack" application, a particle tracking program).

Table 5 For the SPEC 2000 benchmarks and the 1-issue architecture, the expected number of cycles executed by the application ($\times 10^9$) when scheduling is done optimally using perfect profile information (*gold*), and the percentage increase in cycles executed when using (a) perfect profile information (*no error*), (b) imperfect profile information obtained by profiling on a single representative input (*error, 1 input*), and (c) imperfect profile information obtained by profiling on a set of 30 representative inputs (*error, 30 inputs*), for the constraint programming scheduler (*copt*), the list scheduler with the decision tree heuristic (h_{dt}), the list scheduler with Bringmann's heuristic (h_{dhasy}), and the list scheduler with the G* heuristic (h_{g*}). The entries in grey indicate a percentage increase of 2% or greater.

		Profiling											
		no error				*error, 1 input*				*error, 30 inputs*			
application	*gold*	*copt*	h_{dt}	h_{dhasy}	h_{g*}	*copt*	h_{dt}	h_{dhasy}	h_{g*}	*copt*	h_{dt}	h_{dhasy}	h_{g*}
ammp	26,847.1	0.0	0.1	0.2	1.2	0.0	0.1	0.3	1.0	0.0	0.1	0.2	0.8
applu	1,226.1	0.0	0.4	0.6	0.4	0.1	0.5	0.7	0.3	0.0	0.6	0.8	0.2
apsi	4,884.2	0.0	1.1	1.2	1.0	0.3	1.7	2.1	1.6	0.1	1.1	1.2	1.0
art	3,659.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bzip2	20,471.8	0.0	0.1	0.7	0.3	0.1	0.3	0.9	0.6	0.0	0.1	0.6	0.2
crafty	8,406.7	0.0	0.2	1.6	0.4	0.2	0.6	2.1	1.1	0.0	0.2	1.6	0.4
eon	11,898.7	0.0	1.6	3.9	2.0	0.3	2.4	4.2	2.8	0.0	1.3	3.6	1.8
equake	4,113.7	0.0	0.5	0.5	0.5	0.0	0.4	0.4	0.4	0.0	0.4	0.4	0.5
facerec	6,792.4	0.0	0.3	0.6	0.4	0.1	0.4	0.6	0.5	0.0	0.4	0.5	0.4
fma3d	11,075.7	0.0	0.8	3.2	1.0	0.3	1.3	4.0	2.8	0.0	0.7	3.0	1.0
galgel	1,335.6	0.0	0.3	0.3	0.3	0.2	3.8	4.8	2.7	0.0	0.8	2.6	0.3
gap	612,924.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
gcc	6,178.9	0.0	0.1	1.3	0.3	0.1	0.3	1.3	0.7	0.0	0.1	1.2	0.3
gzip	19,451.0	0.0	0.0	0.3	0.2	0.2	0.3	0.6	0.6	0.0	0.1	0.4	0.2
lucas	366.7	0.0	0.0	0.0	0.1	0.0	0.2	0.2	0.2	0.0	0.0	0.0	0.1
mcf	4,844.8	0.0	0.0	1.2	0.1	0.1	0.2	1.2	0.5	0.0	0.0	1.2	0.1
mesa	15,731.8	0.0	0.2	1.1	0.2	0.0	0.3	1.0	0.5	0.0	0.2	0.8	0.2
mgrid	365.9	0.0	0.4	0.5	0.5	0.2	2.6	2.7	1.6	0.0	0.5	0.7	0.5
parser	25,967.8	0.0	0.3	1.9	1.8	0.2	0.5	1.6	1.7	0.0	0.3	1.4	1.2
perlbmk	32,491.2	0.0	0.1	0.5	0.1	0.1	0.4	1.0	0.7	0.0	0.1	0.5	0.1
sixtrack	3,561.5	0.0	3.4	3.5	3.4	0.4	4.8	4.8	5.2	0.0	3.4	3.5	3.4
swim	9.0	0.0	0.2	1.3	0.2	1.1	22.7	24.0	14.2	0.0	17.1	4.8	0.2
twolf	22,997.0	0.0	0.2	3.2	0.3	0.5	1.1	3.3	1.3	0.0	0.2	2.3	0.3
vortex	10,684.6	0.0	0.2	0.4	1.4	0.0	0.3	0.6	1.2	0.0	0.1	0.4	1.4
vpr	13,891.6	0.0	0.2	0.8	0.3	0.0	0.4	0.9	0.7	0.0	0.2	0.7	0.3
wupwise	9,849.5	0.0	0.8	1.8	0.9	0.2	1.6	2.4	1.9	0.1	0.8	1.6	0.9
average		0.0	0.4	1.2	0.7	0.2	1.8	2.5	1.7	0.0	1.1	1.3	0.6
std. dev.		0.0	0.7	1.1	0.8	1.1	7.9	8.2	6.0	0.1	4.9	2.5	0.7

5 Discussion

The discussion of the experimental results are divided according to the three questions we addressed in our study. In Tables 4–6, entries in grey indicate a percentage increase of 2% or greater, as speedups of 2–3% or better are considered significant in profile-directed compiler optimizations[3].

Q1. In the case where the exit probabilities are estimated dynamically, how sensitive are the superblock schedulers to the accuracy of the profile information?

Our experimental results show that the sensitivity of the schedulers to the accuracy of the profile information depends on the width of the architectures, ranging from potentially very sensitive for the 1-issue architecture to insensitive for the 6-issue architecture (see the columns under "Profiling" in Table 4 for a high level summary for all issue widths and see Table 5 for detailed statistics for the 1-issue architecture). For example, consider the list scheduler with the heuristic h_{dhasy} on the 1-issue architecture (Table 5). Given perfect profile information, the average percentage increase from the gold standard across all 26 applications is 1.2% (i.e., nearly optimal) and only four applications exceed the threshold of 2.0% increase or greater. If we profile on a single input, the average percentage increase is 2.5% and, more importantly, the standard deviation is significantly larger and ten of the applications exceed the 2.0% threshold. Finally, if we profile on a representative set of 30 inputs, the performance of the heuristic very nearly approximates the perfect profile case. Hence, the scheduler is sensitive to the accuracy of the profile information and investing more effort into gathering more accurate profiles pays off. Similar, but less dramatic observations hold for the heuristics h_{dt} and h_{g*}.

[3] This is especially true in the case of instruction scheduling which rarely or never leads to decreased performance. Of course, in isolation a small improvement of 2–3% would not be worthwhile. However, instruction scheduling is just one of many optimizations that a compiler performs. Individually, most or all such optimizations will each lead to small improvements—inconsequential in themselves, but cumulatively offering a significant impact on performance.

Table 6 For the SPEC 2000 benchmarks and the 1-issue architecture, the expected number of cycles executed by the application ($\times 10^9$) when scheduling is done optimally using perfect profile information (*gold*), and the percentage increase in cycles executed when using (a) the Trimaran fixed policy $p = 1$, (b) the fixed policy $p = 50$, and (c) the fixed policy $p = 75$, for the constraint programming scheduler (*copt*), the list scheduler with the decision tree heuristic (h_{dt}), the list scheduler with Bringmann's heuristic (h_{dhasy}), and the list scheduler with the G* heuristic (h_{g*}). The entries in grey indicate a percentage increase of 2% or greater.

	Profiling	*No profiling*											
		$p = 1$				$p = 50$				$p = 75$			
application	*gold*	*copt*	h_{dt}	h_{dhasy}	h_{g*}	*copt*	h_{dt}	h_{dhasy}	h_{g*}	*copt*	h_{dt}	h_{dhasy}	h_{g*}
ammp	26,847.1	0.1	0.6	0.7	1.8	0.2	0.0	0.1	0.1	0.1	0.0	0.2	0.2
applu	1,226.1	1.0	1.9	3.3	3.6	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.0
apsi	4,884.2	5.6	8.1	12.8	13.6	0.4	1.2	1.5	1.2	0.3	0.9	1.2	1.2
art	3,659.9	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bzip2	20,471.8	1.0	1.5	1.8	2.6	0.5	0.5	1.0	0.5	0.6	0.6	0.6	0.6
crafty	8,406.7	3.5	4.4	6.3	7.1	0.1	0.3	1.7	0.4	0.3	0.5	0.7	0.6
eon	11,898.7	4.9	7.9	8.6	10.8	1.9	2.4	4.1	2.4	2.4	2.8	3.1	2.9
equake	4,113.7	0.0	0.6	0.6	0.6	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0
facerec	6,792.4	0.4	0.7	1.1	1.4	0.3	0.4	0.6	0.4	0.4	0.4	0.5	0.5
fma3d	11,075.7	6.1	6.4	7.5	15.8	0.2	0.8	3.6	0.8	2.4	2.0	7.6	7.6
galgel	1,335.6	3.2	25.2	25.3	25.5	2.2	3.5	6.7	2.5	2.2	2.5	2.5	2.5
gap	612,924.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
gcc	6,178.9	1.2	1.7	2.0	3.7	0.1	0.2	1.4	0.2	0.6	0.7	1.2	0.7
gzip	19,451.0	1.3	1.4	1.7	2.5	0.4	0.4	0.7	0.4	0.5	0.5	0.5	0.5
lucas	366.7	0.1	1.3	1.3	1.4	1.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
mcf	4,844.8	1.3	1.5	2.0	2.1	0.5	0.5	1.2	0.5	0.5	0.5	0.5	0.5
mesa	15,731.8	1.1	1.4	2.4	2.9	0.0	0.2	1.3	0.2	0.6	0.9	1.5	1.0
mgrid	365.9	1.7	23.4	23.4	23.5	0.7	0.6	0.8	0.6	0.6	0.6	0.6	0.6
parser	25,967.8	1.3	1.9	3.0	5.0	0.4	0.4	0.9	0.4	0.4	0.4	0.4	0.4
perlbmk	32,491.2	1.6	2.2	3.4	4.1	0.2	0.8	1.2	0.8	0.8	0.9	0.9	0.9
sixtrack	3,561.5	8.1	18.8	19.5	19.9	6.3	9.6	6.6	9.6	6.5	9.8	9.8	9.8
swim	9.0	81.0	129.0	131.6	131.8	1.2	74.6	75.8	37.3	1.2	37.3	37.3	37.3
twolf	22,997.0	6.8	8.3	9.3	10.1	0.2	0.4	2.7	0.4	0.4	0.5	0.8	0.6
vortex	10,684.6	0.3	0.6	1.0	2.1	0.5	0.6	0.7	0.6	0.2	0.3	0.4	0.3
vpr	13,891.6	0.9	1.2	1.8	2.7	0.1	0.4	0.8	0.4	0.9	1.0	1.1	1.0
wupwise	9,849.5	3.6	4.7	4.9	7.5	2.6	1.9	3.1	2.1	2.2	2.5	2.7	2.7
average		5.2	9.8	10.6	11.6	0.8	3.8	4.5	2.4	0.9	2.5	2.8	2.8
std. dev.		15.6	25.3	25.7	25.6	1.3	14.6	14.7	7.4	1.4	7.4	7.4	7.4

The above discussion was for the 1-issue architecture. As the issue width increases the sensitivity decreases until for the 6-issue architecture there is no practical difference between the perfect profile information, profiling on a single input, and profiling on a representative set of 30 inputs (not shown). This is important, as it means that for larger issue width architectures putting effort into increasing the profile accuracy, such as by profiling on multiple inputs and combining the profiles into a single hybrid profile, is unnecessary as it does not pay off in terms of better or more robust schedules.

In our discussion of related work (Section 3), we noted that all previous experimental comparisons have made the assumption of perfect profile information and we claimed there that this potentially gives misleading results. If we compare the schedulers using perfect profile information (column "Profiling, no error" in Table 4), it can be seen that the exact scheduler *copt* is only slightly better and that there is little to distinguish the three heuristics h_{dt}, h_{dhasy}, and h_{g*}. However, once realistic profiling error is introduced, it can be seen the heuristic h_{dhasy} could be brittle, whereas the other schedulers were more robust to inaccuracies in the estimates. The exact scheduler *copt* was always better than the list scheduler with various heuristics and was almost completely insensitive to inaccuracies in profiling information. Although the improvements obtained by the exact scheduler do come at a computational price as the time to schedule can be considerably higher (see [28] for details), the exact scheduler is applicable when longer compile times are tolerable, such as when compiling for software libraries, digital signal processing, or embedded applications [18], Alternatively, heuristic approaches have the advantage that they are fast, albeit with some loss of performance in the resulting schedules. Here, the heuristic h_{g*} has much to recommend it as it is fast, relatively accurate, and robust across all architectures. This contradicts current practice as h_{dhasy} is the default heuristic in the Trimaran compiler [6].

Q2. In the case where the exit probabilities are estimated statically, how sensitive are the superblock schedulers to the choice of fixed policy?

In the case where profiling information is not available and the exit probabilities are set statically using a fixed policy, our experimental results show that the performance of the list scheduling algorithm and the exact scheduler are sensitive to the fixed policy. The best policies were to use either $p = 50$ or $p = 75$, where p is the percentage of time that the exit is taken, given that the flow of control has reached this branch. In our experiments, particularly poor choices were $p = 0$ and $p = 1$ (see the columns under "No profiling" in Table 4 for a high level summary for all issue widths and see Table 6 for detailed statistics for the 1-issue architecture). For example, for the heuristic h_{dhasy} on the 1-issue architecture, the average percentage increase from the gold standard across all 26 applications is 10.6% when using the static method $p = 1$ and only 2.8% when using the static method $p = 75$ (see Table 4) and the improvements offered by using $p = 75$ range from more modest improvements for the applications applu, parser, and perlbmk to very significant improvements for the applications apsi, galgel, mgrid, sixtrack, swim, and twolf (see Table 6). Similar observations hold for the other issue widths and for the other scheduling techniques for superblocks (*copt*, h_{dt}, and h_{g*}). Our conclusions contradict a previous study where it was found that $p = 0$ performed better than $p = 50$ [11], although in that study only six integer benchmarks and an idealized architecture were used. Our conclusions also contradict current practice as $p = 1$ is effectively the fixed policy used in the Trimaran compiler [6].

In terms of comparing the heuristic and the exact schedulers under the best fixed policy, the experimental results show that the exact scheduler *copt* was always better, but rarely considerably so, than the list scheduler with various heuristics for the single issue architecture. For example, for $p = 50$, the scheduler *copt* was modestly better on the applications eon and galgel and significantly better on sixtrack and swim (see Table 6). For the larger issue width architectures, the advantage of the exact scheduler *copt* over the list scheduler with a heuristic diminished and it is unclear that the extra cost of the exact scheduler is repaid in terms of sufficiently better performance for these architectures. The choice of best heuristic switched back-and-forth depending on the architecture with the heuristic h_{g*} being somewhat more consistently better, but the differences between the heuristics was small.

Q3. Are dynamic methods for estimating the exit probabilities worth the considerable extra effort or are static methods sufficient?

For the question of whether the extra effort required for dynamic methods (profiling) is paid back in sufficiently better performance, the answer once again depends on the issue width of the architecture. Our experimental results show that for architectures that are single issue, profiling can lead to better schedules than static methods for the list scheduler with the heuristics h_{dt}, h_{dhasy}, and h_{g*} (to see this, compare the column "Profiling, error (30 inputs)" in Table 5 with the column "No profiling, $p = 50$" in Table 6; both of these tables show the detailed data for the 1-issue architecture). Across the three heuristics, the improvements offered by profiling range from more modest improvements for the applications eon, galgel, and wupwise to quite significant improvements for the applications sixtrack and swim. For example, for the heuristic h_{dhasy} the average percentage increase from the gold standard across all 26 applications is 1.3% (i.e., nearly optimal) when profiling using 30 inputs and is 4.5% when using the static method $p = 50$. Since speedups of only 2–3% are considered significant, profiling would be worth the effort for the single issue architecture.

However, for architectures with larger issue widths the extra effort required for dynamic methods does not pay off in noticeably better performance. In our experiments, as long as one of the better static policies was chosen (e.g., $p = 50$), there was no practical difference in performance between dynamic methods and static methods for either the constraint programming scheduler (*copt*) or the list scheduler with the heuristics h_{dt}, h_{dhasy}, and h_{g*} (to see this, compare the columns "Profiling, error" and "No profiling, $p = 50$" for 2-, 4-, and 6-issue in Table 4). Given that profiling can be much more difficult and time consuming than static methods, an important lesson from our experiments is that using static methods does not noticeably degrade performance on larger issue architectures.

Acknowledgements This research was supported by an IBM Center for Advanced Studies (CAS) Fellowship, an NSERC Postgraduate Scholarship, and an NSERC CRD Grant. The experimental results were obtained using the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET). The authors are grateful to the anonymous referees for their comments, which prompted us to re-examine some of our assumptions and to improve the paper.

References

1. ARM. ARM achieves 10 billion processor milestone. Press release, January 2008. Retrieved from http://www.arm.com/about/newsroom/19720.php.
2. ARM. The ARM Cortex-A9 processor. *ARM White Paper*, September 2009. Available at: http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf.
3. P. Berube and J. N. Amaral. Aestimo: A feedback-directed optimization evaluation tool. In *IEEE Int'l Symposium on Performance Analysis of Systems and Software*, pp. 251–260, 2006.
4. P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In *2007 SPEC Workshop*, 2007.
5. R. A. Bringmann. *Enhancing Instruction Level Parallelism through Compiler-Controlled Speculation*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

6. L. N. Chakrapani, J. Gyllenhaal, W. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran: An infrastructure for research in instruction-level parallelism. In *Proc. of the 17th Int'l Workshop on Languages and Compilers for High Performance Computing*, pp. 32–41, 2005.
7. C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with application to superblocks. In *Proc. of the 29th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 58–67, 1996.
8. Y. Chen, H. Yuanjie, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 data sets. In *Proc. of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, pp. 448–459, 2010.
9. T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proc. of the 29th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 36–45, 1996.
10. D. Cormie. The ARM11 microarchitecture. *ARM White Paper*, 2002.
11. B. Deitrich and W. Hwu. Speculative hedge: Regulating compile-time speculation against profile variations. In *Proc. of the 29th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 70–79, 1996.
12. L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *J. Instruction-Level Parallelism*, 5, 2003.
13. A. E. Eichenberger and W. M. Meleis. Balance scheduling: Weighting branch tradeoffs in superblocks. In *Proc. of the 32nd Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 272–283, 1999.
14. J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, C-30:478–490, 1981.
15. J. A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Morgan Kaufmann, 2005.
16. J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proc. of the Fifth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95, 1992.
17. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of the IEEE 4th Annual Workshop on Workload Characterization*, pp. 3–14, 2001.
18. R. Govindarajan. Instruction scheduling. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook*, pp. 631–687. CRC Press, 2003.
19. R. Gupta, E. Mehofer, and Y. Zhang. Profile-guided compiler optimizations. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook*, pp. 143–174. CRC Press, 2003.
20. R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu. Superblock formation using static program analysis. In *Proc. of the 26th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 247–255, 1993.
21. J. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, 1983.
22. J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.
23. W. C. Hsu, H. Chen, P. C. Yew, and D.-Y. Chen. On the predictability of program behavior using different input data sets. In *Proc. of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, pp. 45–53, 2002.
24. 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, 1993.
25. IBM. *Power ISA, Version 2.06 Revision B*, July, 2010. Available at: http://www.power.org/resources/reading/.
26. Intel. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 1: Basic Architecture*, April 2011. Available at: http://www.intel.com/Assets/PDF/manual/253665.pdf.
27. Intel. *Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture*, Revision 2.2, 2006.
28. A. M. Malik, M. Chase, T. Russell, and P. van Beek. An application of constraint programming to superblock instruction scheduling. In *Proc. of the 14th Int'l Conference on Principles and Practice of Constraint Programming*, pp. 97–111, 2008.
29. W. M. Meleis, A. E. Eichenberger, and I. D. Baev. Scheduling superblocks with bound-based branch tradeoffs. *IEEE Trans. on Computers*, 50(8):784–797, 2001.
30. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
31. T. Russell, A. M. Malik, M. Chase, and P. van Beek. Learning heuristics for the superblock instruction scheduling problem. *IEEE Trans. Knowl. Data Eng.*, 21(10):1489–1502, 2009.
32. G. Shobaki and K. Wilken. Optimal superblock scheduling using enumeration. In *Proc. of the 37th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pp. 283–293, 2004.
33. D. W. Wall. Predicting program behavior using real or estimated profiles. WRL Technical Note TN-18, Digital Western Research Laboratory, 1990.
34. Z. Wang and N. Rubin. Evaluating the importance of user-specific profiling. In *Proc. of the 2nd USENIX Windows NT Symposium*, 1998.
35. Y. Wu. Accuracy of profile maintenance in optimizing compilers. In *Proc. of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, pp. 27–38, 2002.