System-Level Performance Metrics for Multiprogram Workloads

Assessing the performance of multiprogram workloads running on multithreaded hardware is difficult because it involves a balance between single-program performance and overall system performance. This article argues for developing multiprogram performance metrics in a top-down fashion starting from system-level objectives. The authors propose two performance metrics: average normalized turnaround time, a user-oriented metric, and system throughput, a system-oriented metric.

Stijn Eyerman
Lieven Eeckhout
Ghent University

Performance metrics are the foundation of experimental computer science and engineering. Researchers and engineers use quantitative metrics for assessing their new ideas and engineering progress. Obviously, adequate metrics are of primary importance to research progress, whereas inappropriate metrics can drive research and development in wrong or unfruitful directions.

The recent trend toward multicore and many-core systems makes adequate performance metrics for assessing the performance of multithreaded computer systems running multiple independent programs essential. There are two key reasons for this need. First, as the number of on-chip cores increases exponentially according to Moore's law, more multiprogram workloads will run on the hardware. Second, coexecuting programs affect each other's performance through sharing of resources such as memory, off-chip bandwidth, and (potentially) on-chip caches and interconnection networks. In addition, programs can share resources within a single simultaneous multithreading (SMT) core.

Quantifying single-program performance is well understood. Researchers have reached the consensus that the performance metric of choice for assessing a single program's performance is its execution time.[1] For single-threaded programs, execution time is proportional to CPI (cycles per instruction) or inversely proportional to IPC (instructions per cycle), provided that cycle time and dynamic instruction count are constant. However, for multithreaded programs, IPC and CPI are poor performance metrics. They can give a skewed performance picture if threads spend time in spin-lock loops (and other synchronization mechanisms) and idle loops in system code without making forward progress. For

Published by the IEEE Computer Society

multithreaded programs, researchers should use total execution time instead.[2]

Quantifying the performance of a computer system executing multiple programs concurrently, on the other hand, is more difficult than for single-program execution because the coexecuting programs affect each other's performance. Multiprogram workload performance is the result of a delicate balance between single-program performance and overall system performance. Researchers have not reached a consensus on how to quantify multiprogram workload performance running on multithreaded hardware, and the issue is still open to debate. For example, some researchers use throughput (the sum of IPCs) as a performance metric; others use the harmonic-mean (hmean) metric; others use weighted speedup; and still others use metrics that assess the fairness level given to each coexecuting program.

In this article, we make the following contributions and insights:

- We advocate developing performance metrics for multiprogram workloads in a top-down fashion by starting from system-level objectives such as program turnaround time and system throughput.
- We identify key multiprogram performance metrics: average normalized turnaround time (*ANTT*)—a user-oriented performance metric—and system throughput (*STP*)—a system-oriented performance metric. *STP* corresponds to the weighted speedup metric proposed by Snavely and Tullsen;[3] and *ANTT* corresponds to the reciprocal of the hmean metric proposed by Luo, Gummaraju, and Franklin.[4] We provide a system-level meaning for both. Another frequently used metric, IPC throughput, has no system-level meaning and should therefore be disregarded.
- We make the case that a comprehensive performance study should report multiple system-level performance metrics. For example, for a general-purpose computer system, *ANTT* and *STP* quantify user-perceived and system-perceived performance, respectively, and combining the two metrics provides insight into the trade-off between single-program performance and overall system performance.
- We generalize a previously proposed fairness metric that assumes equal priority levels for all coexecuting programs in multiprogram workloads by including system-level priorities used by system software.

The multiprogram performance metrics described here will be useful in evaluating design alternatives in system software and architecture in the multicore era.

System-level performance criteria

The criteria for evaluating multiprogram computer systems are based on the user's perspective and the system's perspective.[5] A user-oriented metric quantifies how fast a single program is executed. For example, *turnaround time* quantifies the time between submitting a job and its completion. *Response time* measures the time between submitting a job and receiving its first response; this metric is important for interactive applications. Other user-oriented criteria can quantify whether deadlines are met for soft real-time applications or whether performance isolation or performance predictability can be guaranteed.

A system-oriented metric quantifies how efficiently the system uses resources. For example, *throughput* quantifies the number of programs completed per unit of time. Other system-oriented metrics quantify the processor's busy time, whether fairness is guaranteed among coexecuting programs, how well priorities are enforced, and how well resources are balanced.

A metric's appropriateness depends on the evaluation criterion and the system under evaluation. For example, for systems in a real-time environment, throughput is a lesser concern than whether deadlines are met. But in an interactive environment, response time might be as important as turnaround time and system throughput. Also, assessing multiprogram computer system performance typically involves a delicate and complex balance between

multiple metrics. For example, favoring short-running programs can maximize system throughput, but this would increase the turnaround time of long-running programs and could even starve long-running programs. Instead, a designer might want to optimize system throughput while guaranteeing some maximum dilation of program turnaround time.

We advocate that system software developers and computer architects should derive performance metrics from system-level criteria in a top-down fashion. In particular, we are concerned with performance metrics for general-purpose computer systems that execute multiprogram workloads. The primary performance criteria for a general-purpose system are program turnaround time and system throughput. That is, the design aim of a general-purpose computer system is to complete individual jobs as soon as possible while maximizing overall system throughput. The other performance objectives mentioned can be quantified easily. For example, engineers can quantify soft real-time performance by counting the number of missed deadlines. Likewise, they can evaluate performance isolation by determining whether a program meets a preset performance target irrespective of its coexecuting programs.

System-level performance metrics

Before discussing the user-oriented and system-oriented metrics, we will introduce some terminology and notation.

Terminology and notation

Single-program mode means that a single program has exclusive access to the computer system. It has all system resources at its disposal and is never interrupted or preempted during its execution. *Multiprogram mode* means that multiple programs are coexecuting on the computer system. One example of multiprogram mode is time sharing in a single-threaded computer system. Another example is multiple programs coexecuting on a multicore processor in which each core executes a single program or multiple programs through time sharing. Yet another example is multiple programs coexecuting on an SMT processor.

Because a program's *execution time* is the primary performance metric for both single-threaded and multithreaded workloads, we use it as the foundation of our performance metrics. We denote the number of clock cycles to execute a program i under single- and multiprogram mode as C_i^{SP} and C_i^{MP}, respectively. A program's C_i^{MP} depends on its coexecuting program(s).

Turnaround time

The primary user-oriented performance metric is program turnaround time. We define the normalized turnaround time of program i as

$$NTT_i = \frac{C_i^{MP}}{C_i^{SP}}$$

Normalized turnaround time is a value larger than or equal to 1 and quantifies the user-perceived turnaround time slowdown due to multiprogram execution. Multiprogram execution in which n programs are coexecuting results in an NTT that varies between 1 and n. For example, on single-threaded hardware, NTT equals 1 when executing instructions completely hide the I/O latency of one program from other coexecuting programs. NTT equals n when the coexecuting programs are computation-intensive (no I/O) with fair access to computation resources. In idealized multithreading (no interprogram interference through resource sharing) with n programs running concurrently on n-threaded hardware, NTT equals 1; in practice, however, there is interprogram interference, which results in an NTT greater than 1.

Using per-program normalized turnaround time, we define the average normalized turnaround time across n independent coexecuting programs in multiprogram mode as

$$ANTT = \frac{1}{n}\sum_{i=1}^{n} NTT_i = \frac{1}{n}\sum_{i=1}^{n} \frac{C_i^{MP}}{C_i^{SP}} \quad (1)$$

ANTT is a lower-is-better metric and is computed as the arithmetic average across the individual *NTT*s, following John, who provides a comprehensive discussion on when to use the arithmetic or the harmonic average.[6] For *ANTT*, single-threaded execution time is the baseline we compare against—consistent with a user-oriented view—and thus the appropriate average is the arithmetic average.

If appropriate, we can define related performance metrics such as the maximum normalized turnaround time (*MNTT*) observed across the coexecuting programs as

$$MNTT = \max_{i=1}^{n} \frac{C_i^{MP}}{C_i^{SP}}$$

System throughput

System throughput is the number of programs completed per unit of time. In other words, when optimizing for *STP*, we aim to maximize per-program progress when coexecuting multiple programs in multiprogram mode. We therefore define a program's normalized progress as

$$NP_i = \frac{C_i^{SP}}{C_i^{MP}}$$

NP is the reciprocal of *NTT* and ranges between $1/n$ and 1, with n the number of coexecuting programs in the multiprogram workload. The intuitive understanding is that *NP* quantifies the fraction of single-threaded program execution time reached during multiprogram execution. For example, an *NP* of 0.5 means that a program during a 10-ms time slice in multiprogram mode effectively makes 5 ms of single-program progress, equivalent to executing the program in single-program mode for 5 ms.

The system throughput of a multiprogram execution with n programs is

$$STP = \sum_{i=1}^{n} NP_i = \sum_{i=1}^{n} \frac{C_i^{SP}}{C_i^{MP}} \qquad (2)$$

That is, *STP* is the sum of all *NP*s and is a value in the range of 1 to n; *STP* is a higher-is-better metric. Intuitively speaking, *STP* quantifies accumulated single-program progress under multiprogram execution. For example, in a time-shared multiprogram environment running on single-threaded hardware with n coexecuting programs, *STP* varies between 1 and n. In a single-threaded time-shared multiprogram environment, a program's normalized progress decreases by a factor n with n coexecuting computation-intensive programs without I/O, and *STP* equals 1. I/O-intensive programs, on the other hand, can achieve an *STP* larger than 1. Idealized multithreading in which there is no interaction between n coexecuting programs yields an *STP* of n; in other words, all coexecuting programs make the same progress as under single-program mode. In general, though, multithreading is not ideal, and *STP* is lower than n.

In practice

So far, we have assumed that C_i^{SP} and C_i^{MP} are givens. In practice, however, computing them is not simple. We make a distinction between real hardware and simulation-based experiments.

Hardware experiments. The problem in computing C_i^{SP} and C_i^{MP} arises from two sources. First, a program can exhibit nonhomogeneous behavior during the course of its execution; that is, it can go through execution phases exhibiting different behavioral characteristics.[7] Second, multiprogram execution can lead to different relative program progress rates across different systems.[8] For example, on one processor architecture, program x might make twice as much relative progress as its coexecuting program y, whereas on another processor architecture, both programs might make equal relative progress. The end result is that a different workload executes on each processor architecture. This can skew overall performance results unpredictably and lead to incorrect conclusions.

A solution proposed by Tuck and Tullsen is to reiterate the multiprogram workload until convergence.[9] The basic idea is as follows: Assume that program x runs twice as fast as program y; thus, by the time program x terminates its execution, pro-

gram y will have reached half its execution. Restarting program x and coexecuting it with the second half of y will ensure that program y executes to completion. In addition, Tuck and Tullsen propose using multiple relative starting offsets to exercise multiple interleavings of coexecuting programs.

Simulation experiments. In the preceding discussion, we've assumed that all programs run to completion—that the metrics are based on a program's total execution cycles C_i^{SP} and C_i^{MP}. However, in practice, it might not be feasible to run benchmarks to completion, especially in a simulation setup. Simulating an entire benchmark execution easily takes several weeks, even on today's fastest simulators running on today's fastest hardware. Researchers and designers therefore typically simulate only representative units. Such sampled simulation uses a small number of simulation points or even a single simulation point per benchmark.[7,10–12] A simulation point has a starting point and an end point to define a unit of work. For example, the SimPoint tool selects fairly large simulation points containing on the order of millions of instructions to represent an entire program execution.[7] Alameldeen and Wood describe ways of defining units of work for multithreaded workloads, such as transactions completed.[2]

For sampled simulation, we must adjust the way we compute the *ANTT* and *STP* performance metrics. We must compute the metrics in terms of the number of cycles to complete a representative unit of work instead of the entire program. In the following equations, we assume a single simulation point per benchmark and single-threaded benchmarks for which *CPI* is an accurate performance metric for assessing the performance of a unit of work. However, we can easily extend the computation to multiple simulation points and multithreaded workloads.

We compute the adjusted *ANTT* as

$$ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{CPI_i^{MP}}{CPI_i^{SP}} \quad (3)$$

and the adjusted STP as

$$STP = \sum_{i=1}^{n}\frac{CPI_i^{SP}}{CPI_i^{MP}} \quad (4)$$

with CPI_i^{SP} and CPI_i^{MP} the cycles per instruction achieved for program i during single-program and multiprogram execution, respectively.

The problem of nonhomogeneous behavior as described for entire program executions also applies to simulation points. Vera et al. advocate reiterating over the simulation points until convergence is reached, a methodology similar to that of Tuck and Tullsen for entire program executions.[13] The downside of this approach is that it increases the simulation time requirements on an already tight simulation budget. An alternative, simulation-friendly approach proposed by Van Biesbrouck, Eeckhout, and Calder is to make sure that the simulation points exhibit homogeneous behavior.[14] Homogeneous behavior allows stopping the simulation at any point and yields accurate performance numbers.

Prevalent performance metrics

Computer architecture researchers use various performance metrics for evaluating multiprogram performance, including IPC throughput, average weighted speedup, and hmean. Researchers frequently use the same metrics for multithreaded processor architectures such as SMT and multicore processors running multiprogram workloads composed of single-threaded programs. We now revisit these prevalent metrics and discuss whether and how they relate to system-level performance.

IPC throughput

IPC throughput is defined as the sum of IPCs of the coexecuting programs:

$$throughput = \sum_{i=1}^{n} IPC_i$$

IPC throughput used to be a frequently used performance metric for evaluating multithreaded hardware executing multiprogram workloads. However, the metric

reflects a computer architect's view of throughput, inasmuch as it doesn't have a physical meaning at the system level in terms of program turnaround time or system throughput. Optimizing a system for IPC throughput can lead to unexpected performance numbers at the system level. Researchers have used the IPC throughput metric less frequently in the past few years. The reason is that they understand that IPC throughput doesn't include any notion of fairness, so performance (as quantified by IPC throughput) can be maximized by favoring high-IPC programs. In addition, as Alameldeen and Wood point out, IPC throughput is a false performance metric for multithreaded workloads because it doesn't represent a unit of work completed.[2] This article provides yet another reason not to use IPC throughput: it doesn't have a system-level meaning.

Weighted speedup

In their work on symbiotic job scheduling, Snavely and Tullsen use weighted speedup as a measure of the goodness or speedup of a coschedule.[3] They define weighted speedup as

$$speedup = \sum_{i=1}^{n} \frac{IPC_i^{MP}}{IPC_i^{SP}}$$

with IPC_i^{SP} and IPC_i^{MP} the number of useful instructions executed per cycle of program i under single-program and multiprogram execution, respectively. The intuitive motivation for using IPC as the basis for the speedup metric is that if one job schedule executes more instructions than another in the same time interval, it is more symbiotic and therefore yields better performance. The weighted speedup metric equalizes the contribution of each program in the job mix by normalizing its multiprogram IPC with its single-program IPC. Because IPC is the reciprocal of CPI, it follows that weighted speedup is nothing other than system throughput (*STP*) as we have defined it (Equation 4). Weighted speedup thus also has a physical meaning at the system level; namely, it quantifies the number of jobs completed per unit of time.

Hmean

Luo, Gummaraju, and Franklin propose taking the harmonic average of the per-program IPC speedup numbers IPC_i^{MP}/IPC_i^{SP}, not the arithmetic average as for weighted speedup.[4] They propose the so-called hmean metric:

$$hmean = \frac{n}{\sum_{i=1}^{n} \frac{IPC_i^{SP}}{IPC_i^{MP}}}$$

They base their reasoning not on any system-level performance consideration but on the observation that the harmonic mean tends to result in lower values than the arithmetic average if one or more programs have a lower IPC speedup. They argue that hmean better captures the notion of fairness than weighted speedup.

From the arguments made in this article, it follows that the hmean metric is the reciprocal of the average normalized turnaround time (*ANTT*) metric (Equation 3). In other words, the hmean metric by itself has no system-level meaning but is inversely proportional to the average program turnaround time.

Priorities and fairness

A performance objective that is completely orthogonal to program turnaround time and system throughput is to preserve priorities set by system software. System software uses priorities to discern high-priority from low-priority tasks, to guarantee soft real-time constraints, and to guarantee maximum response times. The intuitive understanding of priorities is that a program should have computation cycles proportional to its relative priority level. For example, for a workload consisting of two programs with priority levels of $P_1 = 4$ and $P_2 = 6$, respectively, the programs should get $4/(4 + 6) = 40$ percent and $6/(4 + 6) = 60$ percent of the computation cycles, respectively. System software typically uses time multiplexing to enforce priorities by assigning more time slices to programs with a higher priority level.

Computer architects typically don't deal with system-level priorities. Instead, they use the notion of fairness. A system is fair if

Table 1. SMT processor configuration.

Parameter	Value
Fetch policy	Icount 2.4
Pipeline depth	14 stages
Shared reorder buffer size	256 entries
Instruction queues	64 entries in integer queue and floating-point queue
Rename registers	100 integer and 100 floating-point
Processor width	4 instructions per cycle
Functional units	4 integer ALUs, 2 load/store units, and 2 floating-point units
Branch misprediction penalty	11 cycles
Branch predictor	2,048-entry gshare
Branch target buffer	256 entries, 4-way set associative
Write buffer	8 entries
L1 instruction cache	64-Kbyte, 2-way, 64-byte lines
L1 data cache	64-Kbyte, 2-way, 64-byte lines
Unified L2 cache	512-Kbyte, 8-way, 64-byte lines
Unified L3 cache	4-Mbyte, 16-way, 64-byte lines
Instruction/data TLB	128/512 entries, fully-associative, 8-Kbyte pages
Cache hierarchy latencies	L2: 11 cycles
	L3: 35 cycles
	Main memory: 350 cycles
Hardware prefetcher	Stride predictor, 8 stream buffers, 8 entries per stream buffer

the coexecuting programs in multiprogram mode experience equal relative progress with respect to single-program mode.[15] Gabor, Weiss, and Mendelson provide a formal definition of fairness as the minimum ratio of relative normalized progress rates of any two programs running on the system[16]:

$$\begin{aligned} fairness &= \min_{i,j}\left(\frac{\left(\frac{IPC_i^{MP}}{IPC_i^{SP}}\right)}{\left(\frac{IPC_j^{MP}}{IPC_j^{SP}}\right)}\right) \\ &= \min_{i,j}\left(\frac{\left(\frac{CPI_i^{SP}}{CPI_i^{MP}}\right)}{\left(\frac{CPI_j^{SP}}{CPI_j^{MP}}\right)}\right) \\ &= \min_{i,j}\frac{NP_i}{NP_j} \end{aligned} \quad (5)$$

Fairness as defined here is a value that ranges between 0 (no fairness, at least one program starves) and 1 (perfect fairness, all programs make equal single-program progress). Fairness thus is a higher-is-better metric.

The fairness metric assumes that all programs have equal priority levels. To generalize this metric to systems with different priority levels, we define a program's *proportional progress* as

$$PP_i = \frac{NP_i}{\left(\frac{P_i}{\sum_{j=1}^{n} P_j}\right)} = \frac{\left(\frac{CPI_i^{SP}}{CPI_i^{MP}}\right)}{\left(\frac{P_i}{\sum_{j=1}^{n} P_j}\right)}$$

The PP metric quantifies how proportional a program's normalized progress (NP_i) is to its relative priority level ($P_i \Big/ \sum_{j=1}^{n} P_j$), with P_i the priority level for program i. Overall system fairness as defined by Gabor, Weiss, and Mendelson can then be generalized to a fairness metric that includes the notion of system-level priorities:

$$fairness = \min_{i,j}\frac{PP_i}{PP_j}$$

This fairness metric is 0 if at least one program starves and 1 if all programs achieve normalized progress rates proportional to their respective relative priority

levels. Obviously, if all programs have the same priority level, this fairness definition degenerates to the aforementioned fairness definition (Equation 5).

This definition of fairness assumes that the multiprogram performance of the program with the highest priority does not equal single-program performance (which is true for most practical situations). Otherwise, any other program (with a lower priority level) that achieves better normalized progress than its relative priority level, (that is, $PP_i > 1$) results in a fairness value smaller than 1. Artificially slowing lower-priority programs to adhere to their priority levels (and to achieve a fairness of 1), doesn't make sense from a user or a system perspective.

Case study: Evaluating SMT fetch policies

On the basis of the arguments made in this article, we advocate that researchers use multiple metrics for characterizing multiprogram system performance. In particular, the combination of *ANTT* and *STP* provides a clear picture of overall system performance as a balance between user-oriented program turnaround time and system-oriented throughput. To illustrate this, we present a case study in which we compare SMT processor fetch policies This study illustrates the general insights that a combined *ANTT-STP* performance characterization can provide.

Although the case study involves user-level single-threaded workloads, it does not affect the general applicability of the multiprogram performance metrics proposed in this article. In fact, an *ANTT-STP* characterization is applicable to evaluation studies involving multithreaded and full-system workloads as well. In this study, we compute the *ANTT* and *STP* metrics using the CPI-based equations 3 and 4; for multithreaded full-system workloads we would simply use the cycle-count-based equations 1 and 2.

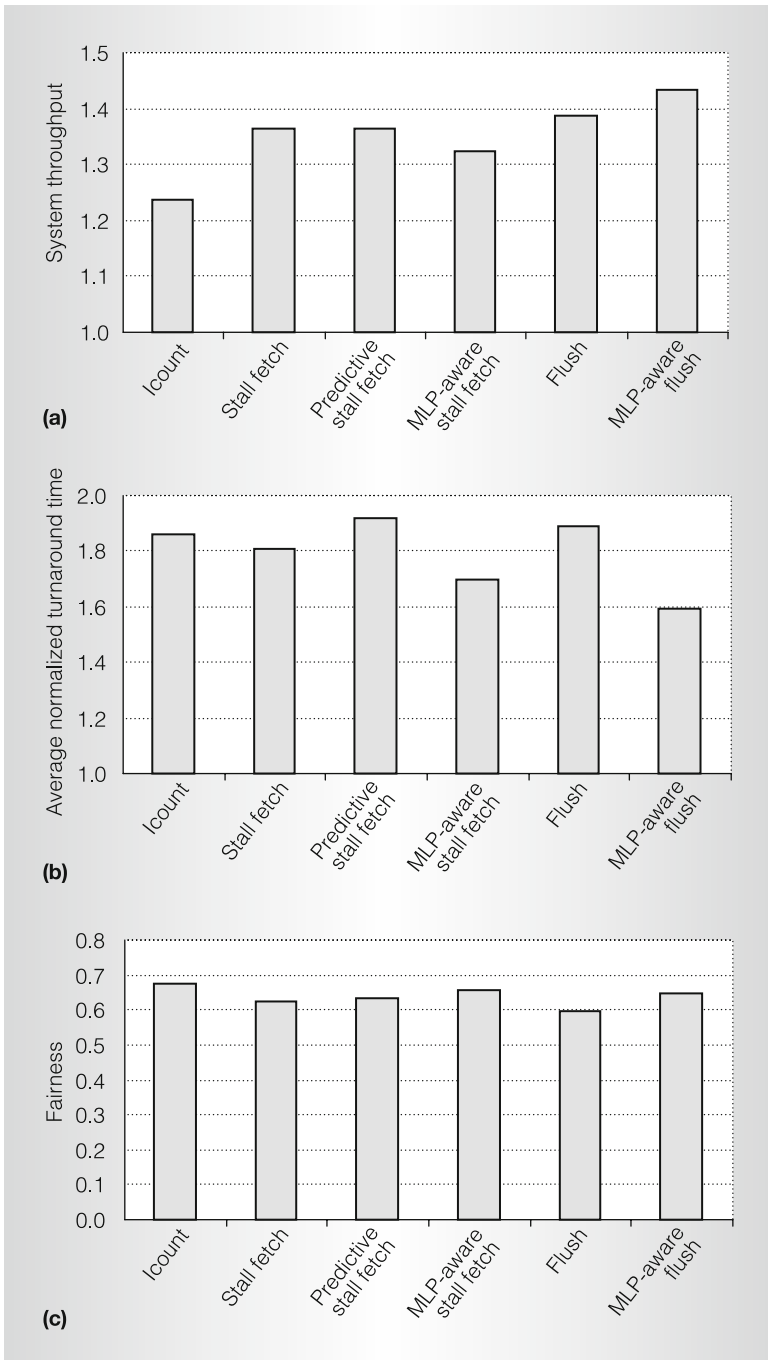

Figure 1. Results for two-program workloads with six SMT processor fetch policies: system throughput (a), average normalized turnaround time (b), and fairness (c).

Setup

In this case study, we compare six SMT fetch policies:

- *Icount* strives to have an equal number of instructions from all coexecuting programs in the front-end pipeline and instruction queues.[17] The following fetch policies extend the Icount policy.
- *Stall fetch* stalls the fetch of a program that experiences a long-latency load

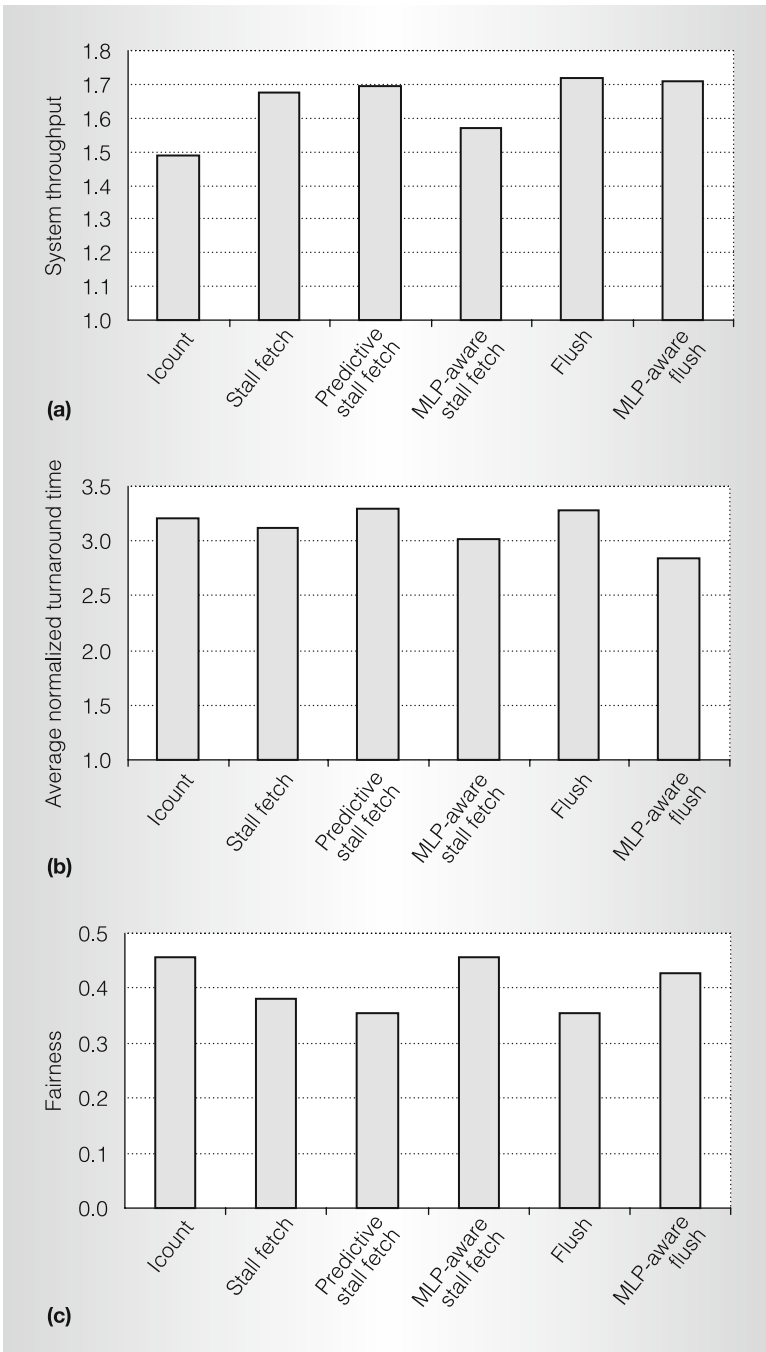

Figure 2. Results for four-program workloads with six SMT processor fetch policies: system throughput (a), average normalized turnaround time (b), and fairness (c).

until the data returns from memory.[18] The idea behind this policy (and the following policies) is to prevent a program experiencing a long-latency load from clogging resources without making forward progress.

- *Predictive stall fetch* extends the stall fetch policy by predicting long-latency loads in the front-end pipeline.[19] Predicted long-latency loads trigger the fetch stalling of a program.
- *MLP-aware stall fetch* predicts long-latency loads and their associated memory-level parallelism (MLP).[20] Specifically, the MLP predictor predicts the MLP distance—the number of instructions in the dynamic instruction stream until the last long-latency load in a burst of long-latency loads. To expose MLP, the policy fetch stalls programs when the number of instructions predicted has been fetched.
- *Flush* flushes on long-latency loads. Our implementation flushes when a long-latency load is detected (this is the TM, or trigger on long-latency miss, described by Tullsen and Brown[18]) and flushes starting from the instruction following the long-latency load (the "next" approach described by Tullsen and Brown).
- *MLP-aware flush* extends the MLP-aware stall fetch policy by flushing instructions if more than m instructions have been fetched since the first in a burst of long-latency loads (with m the MLP distance predicted by the MLP predictor).[20]

All these fetch policies also include the continue-the-oldest-thread mechanism: If all threads stall on a long-latency load, the oldest thread is given priority.[19]

We considered 36 two-program workloads and 30 four-program workloads. We composed these workloads randomly from simulation points (with 200 million instructions each) chosen using SimPoint for the SPEC CPU2000 benchmarks. We considered a four-wide, superscalar, out-of-order SMT processor with an aggressive hardware data prefetcher with eight stream buffers driven by stride predictors. Table 1 (page 48) shows the processor configuration.

Results and discussion

Figures 1 and 2 show *STP* and *ANTT* averaged across a collection of two-program and four-program workloads. In addition,

they show fairness, assuming the coexecuting programs have equal system-level priorities. These graphs show that the MLP-aware flush policy outperforms Icount for both the two- and four-program workloads. That is, it achieves a higher system throughput and a lower average normalized turnaround time, while achieving a comparable fairness level. The same is true when we compare MLP-aware flush against flush for the two-program workloads; for the four-program workloads, MLP-aware flush achieves a much lower normalized turnaround time than flush at a comparable system throughput.

The comparison between predictive stall fetch and MLP-aware stall fetch is less obvious: MLP-aware stall fetch achieves a smaller *ANTT*, whereas predictive stall fetch achieves a higher *STP*. This illustrates the delicate balance between user-oriented and system-oriented views of performance. If user-perceived performance is the primary objective, MLP-aware stall fetch is the better fetch policy. On the other hand, if system-perceived performance is the primary objective, predictive stall fetch is the policy of choice.

Interestingly, making the fetch policy MLP-aware increases the fairness of the predictive stall fetch and flush policies up to the fairness level of the Icount policy. (See the MLP-aware stall fetch and MLP-aware flush policies in Figures 1c and 2c.)

As an illustration of the pitfall in using IPC throughput as a performance metric, compare Figure 3, which shows IPC throughput, with Figure 1a, which shows system throughput (STP) for two-program workloads. Using IPC throughput as a performance metric, you would conclude that the MLP-aware flush policy is comparable to the flush policy. However, it achieves a significantly higher system throughput (STP). Thus, IPC throughput is a potentially misleading performance metric.

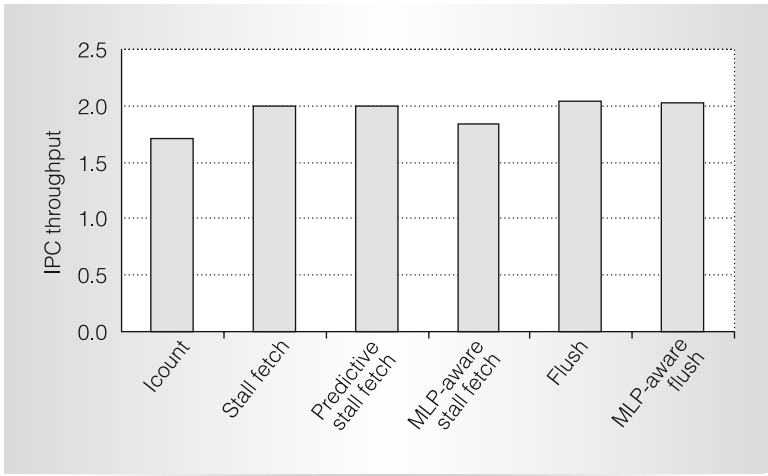

Figure 3. IPC throughput for two-program workloads with six SMT processor fetch policies.

While this article lays the foundation for system-level performance metrics of multiprogram workloads running on multithreaded hardware, there are a number of open research avenues in this area. In particular, creating multiprogram workloads (that is, what programs to combine to form a multiprogram workload?) is challenging. For one, the multiprogram workloads should be diverse and representative—that is, they should cover the workload space. And second, the multiprogram workloads should be simulation friendly—that is, while the number of potential multiprogram workloads grows exponentially with the number of hardware threads, one cannot include all possible combinations of programs because of experimentation time constraints. MICRO

Acknowledgments

Stijn Eyerman is supported through a PhD student fellowship of the Fund for Scientific Research–Flanders (Belgium). Lieven Eeckhout is supported through a postdoctoral fellowship of the Fund for Scientific Research–Flanders, (Belgium).

References

1. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.
2. A.R. Alameldeen and D.A. Wood, ''IPC Considered Harmful for Multiprocessor Workloads,'' *IEEE Micro*, vol. 26, no. 4, July/Aug. 2006, pp. 8-17.
3. A. Snavely and D.M. Tullsen, ''Symbiotic Job Scheduling for Simultaneous Multi-

threading Processor,'' *Proc. 9th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 00), ACM Press, 2000, pp. 234-244.
4. K. Luo, J. Gummaraju, and M. Franklin, ''Balancing Throughput and Fairness in SMT Processors,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 01), IEEE Press, 2001, pp. 164-171.
5. W. Stallings, *Operating Systems: Internals and Design Principles*, 5th ed., Prentice Hall, 2005.
6. L.K. John, ''Aggregating Performance Metrics over a Benchmark Suite,'' *Performance Evaluation and Benchmarking*, L.K. John and L. Eeckhout, eds., (2006), CRC Press, 2006, pp. 47-58.
7. T. Sherwood et al., ''Automatically Characterizing Large Scale Program Behavior,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 45-57.
8. M. Van Biesbrouck, T. Sherwood, and B. Calder, ''A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation,'' *Proc. Int'l Symp. Performance Analysis of Systems and Software* (ISPASS), IEEE CS Press, 2004, pp. 45-56.
9. N. Tuck and D.M. Tullsen, ''Initial Observations of the Simultaneous Multithreading Pentium 4 Processor,'' *Proc. 12th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 03), IEEE CS Press, 2003, pp. 26-34.
10. T.M. Conte, M.A. Hirsch, and K.N. Menezes, ''Reducing State Loss for Effective Trace Sampling of Superscalar Processors,'' *Proc. Int'l Conf. Computer Design* (ICCD 96), IEEE CS Press, 1996, pp. 468-477.
11. R.E. Wunderlich et al., ''SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling,'' *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 84-95.
12. J.J. Yi et al., ''Characterizing and Comparing Prevailing Simulation Techniques,'' *Proc. 11th Int'l Symp. High-Performance Computer Architecture* (HPCA 05), IEEE CS Press, 2005, pp. 266-277.
13. J. Vera et al., ''FAME: Fairly Measuring Multithreaded Architectures,'' *Proc. 16th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 07), IEEE CS Press, 2007, pp. 305-316.
14. M. Van Biesbrouck, L. Eeckhout, and B. Calder, ''Representative Multiprogram Workloads for Multithreaded Processor Simulation,'' *Proc. IEEE 10th Int'l Symp. Workload Characterization* (IISWC 07), 2007, pp. 193-203.
15. F.J. Cazorla et al., ''Predictable Performance in SMT Processors: Synergy between the OS and SMTs,'' *IEEE Trans. Computers*, vol. 55, no. 7, July 2006, pp. 785-799.
16. R. Gabor, S. Weiss, and A. Mendelson, ''Fairness Enforcement in Switch-On Event Multithreading,'' *ACM Trans. Architecture and Code Optimization*, vol. 4, no. 3, Sept. 2007.
17. D.M. Tullsen et al., ''Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor,'' *Proc. 23rd Ann. Int'l Symp. Computer Architecture* (ISCA 96), IEEE CS Press, 1996, pp. 191-202.
18. D.M. Tullsen and J.A. Brown, ''Handling Long-Latency Loads in a Simultaneous Multithreading Processor,'' *Proc. 34th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 01), IEEE CS Press, 2001, pp. 318-327.
19. F.J. Cazorla et al., ''Optimizing Long-Latency-Load-Aware Fetch Policies for SMT Processors,'' *Int'l J. High Performance Computing and Networking*, vol. 2, no. 1, 2004, pp. 45-54.
20. S. Eyerman and L. Eeckhout, ''A Memory-Level Parallelism Aware Fetch Policy for SMT Processors,'' *Proc. 13th Int'l Symp. High-Performance Computer Architecture* (HPCA 07), IEEE CS Press, 2007, pp. 240-249.

Stijn Eyerman is a PhD student in the Electronics and Information Systems Department at Ghent University, Belgium. His research interests include computer architecture in general and performance analysis and modeling in particular. Eyerman has an MS in computer science and engineering from Ghent University.

Lieven Eeckhout is an assistant professor in the Electronics and Information Systems Department at Ghent University, Belgium. His research interests include computer architecture, virtual machines, performance analysis and modeling, and workload characterization. Eeckhout has a PhD in computer science and engineering from Ghent University. He is a member of the IEEE and the ACM.

Direct questions and comments about this article to Lieven Eeckhout, ELIS Dept., Ghent University, Sint-Pieternieuwstraat 41, B-9000 Ghent, Belgium; leeckhou@elis.UGent.be.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/csdl.