Predicting Performance on SMPs.

A Case Study: The SGI Power Challenge*

Nancy M. Amato† Jack Perdue† Andrea Pietracaprina‡
amato@cs.tamu.edu jkp2866@cs.tamu.edu andrea@artemide.dei.unipd.it

Geppino Pucci‡ Mark Mathis†
geppo@artemide.dei.unipd.it mmathis@cs.tamu.edu

Technical Report 99-020
Department of Computer Science
Texas A&M University
October 4, 1999

Abstract

In this work we study the issue of performance prediction on the SGI-Power Challenge, a typical representative of the class of shared-memory Symmetric MultiProcessors. On such a platform, the cost of memory accesses varies depending on their locality and on contention among processors. By running a carefully designed suite of microbenchmarks, we provide quantitative evidence that the interaction with the memory hierarchy affects performance far more substantially than other phenomena related to contention. We also fit three cost functions based on variants of the BSP model, which do not account for the hierarchy, and a newly defined function F, expressed in terms of hardware counters, which captures both memory hierarchy and contention effects. We test the accuracy of all the functions on both synthetic and application benchmarks showing that, unlike the other functions, F achieves an excellent level of predictivity in all cases. Although hardware counters are only available at run time, we give evidence that function F can still be employed as a prediction tool by extrapolating values of the counters from pilot runs on small input sizes.

*This research was supported in part by NATO CRG 961243 "Bulk Synchronous Computational Geometry," and by NSCA grant CCR970010N. The work at Texas A&M was also supported by the NSF by CAREER award CCR-9624315 and grants IRI-9619850, ACI-9872126, EIA-9805823, EIA-9810937, by DOE ASCI ASAP (Level 2 Program) grant B347886, and by the Texas Higher Education Coordinating Board grant ARP-036327-017. Perdue and Mathis supported in part by Dept. of Education Graduate Fellowships.

†Department of Computer Science, Texas A&M University, College Station, TX, USA.
E-mail: {amato,jkp2866,mmathis}@cs.tamu.edu

‡Dipartimento di Elettronica e Informatica, Università di Padova, Italy.
E-mail: {andrea,geppo}@artemide.dei.unipd.it

1 Introduction

Despite the vast body of ingenious parallel algorithmic techniques developed over the last two decades, the widespread use of parallel computers is still hampered by the difficulty of exploiting their massive computational potential to an extent that warrants their large cost. Indeed, it has often been noted that theoretically efficient designs exhibit poor performance when implemented on real machines. Very often, this is due to the inadequacies of the cost functions employed to predict performance, which do not properly account – or totally disregard – aspects of the machine that have a major impact on performance.

Although much progress has been made, the development of adequate tools for predicting actual performance on real machines remains one of the most challenging problems in parallel processing. We believe that further progress towards this goal requires a tighter coupling of cost models to architectures than has been previously employed.

The issue of predictivity is especially challenging for the class of *Symmetric MultiProcessors (SMPs)*. These widely spread parallel platforms are built upon powerful off-the-shelf microprocessors interacting through a distributed shared-memory via a communication medium, typically a bus. In such a system, the cost of an access to a shared datum may vary dramatically: from a few cycles if the data is in first-level cache (L1), to tens of cycles for second-level (L2) cache, to hundreds of cycles if the data must be accessed from main memory. The cost may be even greater in the presence of high contention among the processors for the bus or memory banks, or of (false) data sharing.

Our Contribution In this paper, we study the relative impact on performance of hierarchy and contention phenomena on an SGI-Power Challenge (SGI-PC), which is a typical representative of the class of SMPs. More specifically, we present a suite of synthetic microbenchmarks which exercise different usages of the hierarchy under a set of controlled scenarios obtained by varying the level and type of contention among the processors. Based on the access times measured through the microbenchmarks, we infer parameter values for a set of linear cost functions inspired by some variants of the popular *Bulk-Synchronous Parallel* (*BSP*) model [Val90], and of a newly defined function which relies on the MIPS R10000 hardware counters describing the memory hierarchy usage of a program. While the BSP-derived functions account for bus and bank contention and for data sharing but disregard hierarchy effects, the counter-based function tries to encompass all of these phenomena.

We test the accuracy of the cost functions on the whole set of microbenchmarks, and on application benchmarks, namely, the NAS suite of parallel benchmarks and three sorting algorithms. Our tests show that the function based on hardware counters, which accounts for the crucial impact of the memory hierarchy on performance, achieves an excellent level of predictivity in all cases, with predicted times less than a factor 2 away from actual times on average, and less than a factor 3 in the worst case. In contrast, the cost functions inspired by the BSP-like models, which disregard hierarchy effects, provide performance predictions that can be more than two orders of magnitude away from actual times. Our study provides quantitative evidence that the memory hierarchy is the primary factor that affects performance on SMPs, while the impact of the other phenomena, although noticeable,

is considerably less crucial.

It has to be remarked that while the BSP-like cost functions are easily computed *a priori* by code inspection, the counter-based function involves quantities (such as the number of accesses at the various levels of the memory hierarchy) that are easily computable only at run-time. However, we claim that the latter function can still be used as a prediction tool whenever accurate guesses of such quantities can be inferred from the code or extrapolated from pilot runs on small input sizes. In order to validate our claim, we provide examples of performance prediction for large sorting instances based on extrapolation of the relevant counters.

The counter-based function may also prove useful in the design of software systems, compilers, or large applications, to profile the memory hierarchy usage of critical portions of their code.

Previous Work The issue of performance prediction of parallel software has received considerable attention over the last decade, often within the context of the more general quest for a bridging model of parallel computation, i.e., one that balances among conflicting requirements such as simplicity, accuracy and generality. One of the most popular attempts at defining a bridging model has been made by Valiant [Val90] with the BSP model. BSP is a bulk-synchronous model where computation is organized as a sequence of *supersteps* separated by barrier synchronizations, and processors operate asynchronously within each superstep. The large body of work this model has generated has demonstrated its suitability for the development of portable software (see e.g., [GLR+99]).

Although the original BSP is meant to model distributed-memory architectures, where communication is realized via message passing, two BSP variants specifically tailored to shared-memory systems have been recently developed, namely, the *Queuing Shared Memory* (*QSM*) [GMR99] and the (d,x)-BSP [BGMZ97], which both embody some aspects of memory contention. In particular, QSM's cost function includes a parameter that accounts for the maximum number of concurrent accesses to the same memory location, while (d,x)-BSP's cost function accounts for memory bank contention (parameters d and x represent, respectively, bank delay and banks to processors ratio). The set of BSP-derived cost functions considered in this paper include those of QSM and (d,x)-BSP, and a third function, inspired by the *Extended BSP* (*EBSP*) model [JW98], which extends BSP to account for unbalanced communication. Although EBSP was meant to model message-passing systems, we obtain the cost function by reinterpreting its original definition for a shared-memory machine.

In [ACF90] the *Parallel Memory Hierarchy* (*PMH*) model is introduced which uses a single mechanism to model both interprocessor communication and memory hierarchy in a parallel computer through a tree-structured view of the machine's organization. Although the model encompasses parameters which characterize the performance at each level of the tree, it does not provide a global cost function that can be used to predict program performance.

Finally, hardware counters are nowadays extensively used to profile sequential and parallel code. Examples of such use of the counters on the SGI-PC can be found in [ZLTI96].

2 Hardware and Software Platforms

We conducted our study on an SGI Power Challenge (SGI-PC), a typical representative for the class of SMPs. The SGI-PC configuration we used consists of eight R10000 194 MHz processors, each provided with a 32 KB on-chip instruction cache, a 32 KB on-chip level-1 (L1) data cache, and a 1 MB off-chip unified (instructions and data) level-2 (L2) cache. Cache line size is 32 B (8 words) for L1 and 128 (32 words) for L2. An 8-way interleaved, 2 GB main memory distributed across 8 banks is accessed by the processors through a 1.2GB/s shared-bus using a cache-coherent protocol [Sil95].

All our experiments on the SGI-PC have been coded according to an SPMD bulk-synchronous programming style [Val90, GMR99], where all processors execute the same program consisting of a sequence of *supersteps* separated by barriers. In a superstep, each processor performs a number of *memory accesses* (`load` or `store` instructions) on words which may reside either in the processor's L1/L2 caches or in main memory, and a number of *local operations* on data held in registers. Barriers have been implemented using the SGI native `m_sync()` primitive. In this work, we are mainly interested in predicting the cost of memory accesses and we will not deal with local operations.

The running time of each superstep was measured by mapping the CPU cycle counter to memory (`syssgi()` and `mmap()`), reading that value as each superstep started and ended, and using a scaling factor (`syssgi()` provided) to convert clock cycles to microseconds. Also, in each superstep we monitored loads/stores issued, and primary/secondary cache misses/writebacks at each individual processor by means of some hardware counters provided by the R10000 design [Sil97]. (The employed counters are described in Section 4.2.)

3 Experimental Testbed

In this section, we describe a suite of simple microbenchmarks whose purpose is to measure the cost of accessing the SGI-PC memory system under a variety of scenarios. The suite was designed with the intention of ascertaining the relative impact of the following key phenomena on access time:

Locality: the level of the hierarchy where the accesses take place (i.e., either L1/L2 caches or main memory);

Bus Contention: the volume of bus traffic generated by the accesses;

Bank Contention: the amount of accesses directed to the same memory bank;

Coherence: the different coherency activities triggered by the accesses.

In the generic microbenchmark, x processors, $1 \leq x \leq 8$, perform a given sequence of accesses (either a `load` or a `store` sequence) to a sub-array of a large shared array. The microbenchmark is repeated a number of times to filter out noise in the measurements and cold-start effects. In order to exercise different combinations of the four phenomena illustrated above, we instantiate this generic microbenchmark by varying the number of active processors, the access stride, and the size and base address of the sub-arrays accessed

by the individual processors (different base addresses are used in combination with a given stride to direct accesses to the desired banks). More specifically, we have experiments with 1, 2, 4, 8 processors active, strides ranging from 1 to 256 words, accesses involving 0, 1, 2, 4, 8 distinct banks, and, finally, processors working either exclusively on distinct sub-arrays or concurrently on a coincident sub-array.

Varying the number of active processors allows us to vary the amount of traffic sustained by the bus, and, consequently, to evaluate the impact of bus contention on access times. Varying the stride allows us to measure the impact of both locality and bank contention. To see the latter, consider, for instance, microbenchmarks with a single processor active. When accessing a sub-array of `size(L1)` words with stride 1, after the first iteration of the microbenchmark, all data reside in L1, hence all future accesses will take place in L1. Similarly, accessing a sub-array of `size(L2)` words with stride 8 implies that each access results in an L1 miss and an L2 hit. Finally, accessing an array of $2 \times$ `size(L2)` words with stride $32 * i$, implies that all accesses generate L1 and L2 misses and reach $8/i$ memory banks, for $i = 1, 2, 4, 8$. In what follows, we define the *BanK Contention* (BKC, for short) of a microbenchmark as the ratio of the number of active processors to the number of distinct banks accessed. In case all accesses take place in cache, we set BKC=0. When accesses reach main memory, BKC varies from 1/8 (a single processor accessing all banks) to 8 (all processors accessing the same bank). Clearly, a higher value of BKC corresponds to a higher bank contention.

Table 1 lists the microbenchmarks that make up our suite. All the microbenchmarks have been performed once for loads and once for stores, and by having processors operate once exclusively and once concurrently on sub-arrays, for a total of 4 combinations. Each table entry contains a three-field mnemonic code px.sy.bz identifying the microbenchmark, where x is the number of active processors, y is the stride, and z is the value of BKC. When referring to one of the microbenchmarks for a given combination of access type (either Load (L) or Store (S)) and concurrency policy (either Exclusive (E) or Concurrent (C)), we append the mnemonic code with an extra field identifying the combination. For instance, code p8.s32.b1.LC identifies the microbenchmark where eight processors issue loads to one every 32 words of the same subarray, thus addressing all banks.

Figure 1 provides a graphical representation of the access times measured by running the four combinations of the suite. Specifically, in Figure 1(a) we plot the running time of a selected subset of load and store microbenchmarks as a function of BKC for both the Exclusive and the Concurrent scenario. Figure 1(b) shows a similar plot as a function of the number of active processors, when BKC is fixed to 1. We note that for loads the impact of both bus and bank contention is somewhat minor, while for stores it is more significant, especially when combined with coherency traffic (Concurrent scenario). However, in any case neither of these phenomena affects access times by more than a factor 4, while, as we will see in the following sections, access times may vary by more than two orders of magnitude due to memory hierarchy effects.

4 Predicting Performance

In this section, we introduce a number of cost functions which can be used to predict the running time of a superstep on the SGI-PC. As mentioned before, we will focus on the contribution of memory accesses to the running time. In the first subsection, we present three functions inspired by the BSP variants mentioned in the introduction. These functions base their predictions on quantities that are computable, to a large extent, by inspecting the (assembly) code, and cannot account for memory hierarchy effects. In fact, the code allows one only to distinguish between operations on data held in registers, which we regard as local operations, and accesses to the rest of the memory system, without explicitly distinguishing between accesses to L1/L2 caches or main memory. Hence, the three BSP-like functions considered here treat all such accesses in the same way. In the other subsection, a new function is defined which explicitly accounts for the memory hierarchy and is expressed in terms of the values of the MIPS R10000 hardware counters.

4.1 BSP-like cost functions

We define three cost functions based, respectively, on the QSM, E-BSP, and (d, x)-BSP models. The functions, which are described below, are used to predict the running time of a superstep. In all of the functions H represents the maximum number of memory accesses performed by a processor in the superstep.

Function QSM Function QSM is defined as

$$\max\{g1_{\mathrm{QSM}} \cdot H,\ g2_{\mathrm{QSM}} \cdot K\},$$

where K is the maximum number of accesses performed to the same word by all processors, $g1_{\mathrm{QSM}}$ is the cost per access experienced by a processor, and $g2_{\mathrm{QSM}}$ is the cost per access relative to a single word. According to the model, the second term is expected to dominate in case of high contention at a cell.

Since the function does not distinguish among accesses placed at different levels of the hierarchy, nor does it distinguish among loads and stores, which generally have different costs, one cannot provide unique values for the parameters but, rather, intervals of possible values. On the SGI-PC, we employed the suite of microbenchmarks of Section 3 to obtain minimum and maximum values for each parameter involved in the function, by considering best case and worst case scenarios with respect to phenomena not captured by the parameter.

For $g1_{\mathrm{QSM}}$, the minimum value[1] (0.0083) resulted from the microbenchmark performing loads from L1 (p1.s1.b0.LE), while the maximum (5.35) resulted from the microbenchmark performing stores to main memory with maximum bank and bus contention, and maximum coherency traffic (p8.s256.b8.SC). For $g2_{\mathrm{QSM}}$, the minimum value (0.001) resulted from the microbenchmark with all processors repeatedly loading the same word (a variant of p8.s1.b0.LC), while the maximum (0.67) resulted from the microbenchmark with all processors repeatedly storing the same word (a variant of p8.s256.b8.SC).

[1] All parameter values are expressed in μsec per access.

Hence, we can define an "optimistic" version (QSM.MIN) and a "pessimistic" version (QSM.MAX) of the QSM function based, respectively, on the minimum and maximum values of $g1_{\mathrm{QSM}}$ and $g2_{\mathrm{QSM}}$. The two versions are:

$$\begin{aligned} \text{QSM.MIN} &= \max\{0.0083 \cdot H,\ 0.001 \cdot K\}, \\ \text{QSM.MAX} &= \max\{5.35 \cdot H,\ 0.67 \cdot K\}. \end{aligned}$$

Function EBSP Function EBSP is defined as

$$\max\{g1_{\mathrm{EBSP}} \cdot H,\ g2_{\mathrm{EBSP}} \cdot M/p\},$$

where M is the total number of accesses performed by all p processors, $g1_{\mathrm{EBSP}}$ is the cost per access experienced by a processor when no other processor accesses memory, and $g2_{\mathrm{EBSP}}$ is the cost per access experienced by a processor when all other processors perform approximately the same number of accesses, thus resulting in high bus traffic. Clearly, we expect $g1_{\mathrm{EBSP}} \leq g2_{\mathrm{EBSP}}$. According to the EBSP philosophy, the running time of a superstep characterized by an unbalanced access pattern, i.e., one with $H \gg M/p$, is determined by the time taken by the processor performing the largest number of accesses, as if it worked in isolation, hence the term $g1_{\mathrm{EBSP}} \cdot H$ in the function dominates, whereas when $H \simeq M/p$ the high traffic may slow the processors down, hence the term $g2_{\mathrm{EBSP}} \cdot M/p$ dominates.

As before, we determine minimum and maximum values of the parameters. For $g1_{\mathrm{EBSP}}$, the minimum value (0.0083) resulted from loads from L1 (p1.s1.b0.LE), while the maximum (1.31) resulted from the microbenchmark with one processor accessing main memory with maximum bank contention (p1.s256.b1.SE). For $g2_{\mathrm{EBSP}}$, the minimum value (0.0083) resulted from the microbenchmark with all processors accessing their L1's (p8.s1.b0.LE), while the maximum (5.35) resulted from the microbenchmark with all processors performing the same number of accesses to main memory with maximum bank and bus contention and maximum coherency traffic (p8.s256.b8.SC).

The corresponding "optimistic" and "pessimistic" versions of function EBSP are:

$$\begin{aligned} \text{EBSP.MIN} &= \max\{0.0083 \cdot H,\ 0.0083 \cdot M/p\}, \\ \text{EBSP.MAX} &= \max\{1.31 \cdot H,\ 5.35 \cdot M/p\}. \end{aligned}$$

Function DXBSP Function DXBSP is defined as

$$\max\{g1_{\mathrm{DXBSP}} \cdot H,\ g2_{\mathrm{DXBSP}} \cdot M_b\},$$

where M_b is the total number of accesses directed to the same bank, $g1_{\mathrm{DXBSP}}$ is the cost per access experienced by a processor when bank contention is low, and $g2_{\mathrm{DXBSP}}$ is the cost per access at a bank experienced in case of high bank contention. According to the (d, x)-BSP model, the second term is expected to dominate only when many accesses hit the same bank, which becomes a bottleneck. It has to be remarked that the DXBSP function is difficult to apply since determining the value of M_b for a superstep requires a very precise knowledge of the memory map.

For $g1_{\mathrm{DXBSP}}$, the minimum value (0.0083) resulted from one processor loading from L1 (p1.s1.b0.LE), while the maximum (3.40) resulted from the microbenchmark with all processors accessing main memory with moderate bank contention, BKC = 1, but high bus contention and coherency traffic (p8.s32.b1.SC). For $g2_{\mathrm{DXBSP}}$, the minimum value (0.26) and maximum value (0.33) resulted from microbenchmarks with all processors performing, respectively, loads and stores on distinct words in the same bank (p8.s256.b8.LE and p8.s256.b8.SE). Here, we chose microbenchmarks without concurrent accesses to make sure that the running time was indeed dominated by bank delay and not by other factors (e.g., coherency activities).

The corresponding "optimistic" and "pessimistic" versions of function DXBSP are:

$$\begin{aligned} \text{DXBSP.MIN} &= \max\{0.0083 \cdot H,\ 0.26 \cdot M_b\}, \\ \text{DXBSP.MAX} &= \max\{3.40 \cdot H,\ 0.33 \cdot M_b\}. \end{aligned}$$

4.2 A cost function based on hardware counters

As will be clearly shown by the validations reported in the next section, the accuracy of all of the above BSP-like functions is strongly limited by the fact that they disregard memory hierarchy, which is the main reason for the high variance in the parameter values, and, consequently, for the large gap between the optimistic and pessimistic versions of these functions.

We define a new function F that predicts performance based on the following MIPS R10000 hardware counters:

C0E2	`Loads` issued
C0E3	`Stores` issued
C1E9	L1 misses
C1EA	L2 misses
C1E6	L1 lines written back from L1 to L2
C0E7	Quadwords (16 B) written back from L2 to main mem

The counters provide a detailed account of the memory hierarchy usage. Function F is defined under the assumption that the running time of a superstep is determined by one of the following factors: (1) the accesses issued by some processor at the various levels of the hierarchy (2) the traffic on the bus caused by accesses to main memory; (3) bank contention cause by accesses targeting the same bank. To reflect this assumption, F takes the maximum of three functions F1, F2 and F3 defined as follows. Let p_i and b_j denote, respectively, the i-th processor and the j-th memory bank, with $0 \leq i, j < p$.

$$\begin{aligned} F1 &= \max_{0 \leq i < p} (g1_{\mathrm{F1}} \cdot (\mathrm{C0E2}(p_i) + \mathrm{C0E3}(p_i)) + g2_{\mathrm{F1}} \cdot \mathrm{C1E9}(p_i) + g3_{\mathrm{F1}} \cdot \mathrm{C1EA}(p_i) + \\ &\qquad g4_{\mathrm{F1}} \cdot \mathrm{C1E6}(p_i) + g5_{\mathrm{F1}} \cdot \mathrm{C0E7}(p_i)) \\ F2 &= g1_{\mathrm{F2}} \cdot \sum_{0 \leq i < p} \frac{\mathrm{C1EA}(p_i)}{p} + g2_{\mathrm{F2}} \cdot \sum_{0 \leq i < p} \frac{\mathrm{C0E7}(p_i)}{p} \\ F3 &= \max_{0 \leq j < p} (g1_{\mathrm{F3}} \cdot \mathrm{LM}(b_j) + g2_{\mathrm{F3}} \cdot \mathrm{WB}(b_j)), \end{aligned}$$

where counters C0E2(p_i), C0E3(p_i), C1E9(p_i), C1EA(p_i) and C0E7(p_i) denote the values of the respective counters of processor p_i, while LM(b_j) and WB(b_j), which must be inferred from counters C1EA and C0E7 and from code inspection, denote, respectively, the total number of L2 misses served by bank b_j and the quadword writebacks from L2 to bank b_j.

F1 accounts for the usage of the memory hierarchy by individual processors. Specifically, $g1_{\mathrm{F1}}$ reflects the cost of accessing L1 and is multiplied by the total number of loads and stores made by a processor, since all of them act eventually on L1; $g2_{\mathrm{F1}}$ and $g4_{\mathrm{F1}}$ account for the costs of data movements (respectively misses and writebacks) between L1 and L2; analogously, $g3_{\mathrm{F1}}$ and $g5_{\mathrm{F1}}$ account for the costs of data movements between L2 and main memory. F2 accounts for bus contention and should dominate when most of the processors are active and each active processor issues many requests for data residing in main memory. If, in addition to this condition, most of the access requests are directed to the same bank, then F3, which accounts for bank contention, will dominate.

The table below shows the values of the parameters obtained by fitting each function (through least squares fitting) on a set of microbenchmarks where that function is expected to dominate. Each row of the table reports the parameters of a distinct function (subscripts are omitted for simplicity) and the microbenchmarks used for the fit ('-' is used as a "don't care").

Function	$g1$	$g2$	$g3$	$g4$	$g5$	Microbenchmarks
F1	0.0088	0.055	0.97	0.013	0.026	p1.s1.b0.-E, p1.s8.b0.-E, p1.s32.b1/8.-E
F2	1.17	0.18				p8.s256.b1.-E
F3	0.26	0.051				p8.s256.b8.-E

Although F cannot be immediately computed by simply inspecting the code, it may still be used to predict performance of an application in those situations where accurate estimates of the relevant counters can be inferred *a priori* or extrapolated from pilot runs on small input sizes. An example of such use is given in Section 5.4. Moreover, the function could prove useful in the design of software systems, compilers, or large applications, to profile the memory hierarchy usage of critical portions of their code.

5 Validations

In this section we investigate the predictive quality of the cost functions introduced above. We check the accuracy of the functions over a set of synthetic access patterns, including the suite of microbenchmarks described in Section 3, and on a number of real applications, namely, the NAS Parallel Benchmarks [B+91, B+95], and three bulk-synchronous implementations of parallel sorting.

5.1 Synthetic Patterns

The cost functions have been tested on a large number of synthetic access patterns, obtained by running the original set of microbenchmarks listed in Table 1 under a variety of scenarios

featuring different phenomena that could have an impact on access time. More specifically, along with the already discussed Load/Store–Exclusive/Concurrent combinations, we introduced other variants where processors access data which are present as *clean*, *shared* or *dirty* in some other processors' caches, so to include patterns exercising different aspects of the coherency protocol. Overall, we obtained a *Validation Suite* (VS, for short) of 412 different access patterns.

The table below reports the average and maximum errors incurred by the functions over the entire VS, where the error is computed, for each experiment, as

$$E = \frac{\max\{T, P\}}{\min\{T, P\}},$$

with T (resp., P) denoting the measured (resp., predicted) time of the experiment. Clearly, a value E indicates that the predicted time is $(E - 1) * 100\%$ smaller or larger than the measured time. Figure 2 shows plots of actual and predicted running times for the subset of experiments of Figure 1.(a).

Function	AVG ERR	MAX ERR
QSM-MIN	24.15	88.02
QSM-MAX	53.85	636.79
EBSP-MIN	24.15	88.02
EBSP-MAX	27.29	648.35
DXBSP-MIN	6.36	31.84
DXBSP-MAX	34.8	411.46
F	1.19	1.91

The results of the validations clearly show that disregarding hierarchy effects when evaluating performance has a huge negative impact on predictive accuracy, while modeling other architectural aspects, such as bus and bank contention, or concurrency to the same cell, yields a rather modest payoff in achieving higher accuracy. Finally, the fact that the F function exhibits a high predictive quality on all the experiments in VS suggests that phenomena that were disregarded when defining the function (such as some types of coherency overheads) have only a minor impact on performance.

5.2 Sorting Programs

Our first set of applications consists of three sorting algorithms: *samplesort* [WS88], *columnsort* [L85], and a parallel version of *radixsort* [BLM+98]. These algorithms were chosen because they are well-understood parallel algorithms that exhibit a variety of communication patterns. We coded all algorithms in a bulk-synchronous fashion, with no special effort made to optimize the implementations, since our goal was to test the accuracy of the cost functions rather than to develop efficient algorithms.

Each algorithm was run on a wide range of input sizes; namely, $n/p = 10^5 \cdot i$, for $1 \leq i \leq 10$, where n is the total number of keys to be sorted. For each run, we applied the functions to every superstep. For the most part, applying the functions was straightforward.

The only difficulty was posed by functions DXBSP and F3, which require knowledge of the maximum number of L2 misses and writebacks targeting a particular memory bank. While this information is known for the synthetic access patterns of VS, it cannot be readily obtained for real applications. However, for the purposes of our validations, we have used estimates which assume that all such accesses are equally distributed among the eight banks (i.e., BKC = 1 for our eight-processor system). This is a reasonable assumption for the sorting programs and the NAS benchmarks whose access patterns tend to be balanced amongst the 8 banks.

A summary of our results is contained in Table 2 where we report the average and maximum errors, computed as previously described, for each superstep over all runs. Since the sorting algorithms exhibit a high degree of locality, we would expect the optimistic versions of the BSP-like functions to perform much better than their pessimistic counterparts, and indeed this is the case (errors are not shown for EBSP_{min} and $\text{DXBSP}_{\text{min}}$ because they were almost identical to the errors for QSM_{min}). Although the difference is not as dramatic as for the synthetic applications, F is still clearly seen to be significantly more accurate than any of the BSP-like functions. This indicates that disregarding hierarchy effects results in a significant lack of accuracy even for regular programs.

5.3 NAS Parallel Benchmarks

We next applied the functions to the *NAS Parallel Benchmarks* (*NPB*) [B+91, B+95]. NPB is a set of 8 programs, derived from computational fluid dynamics (CFD) applications, that is designed to evaluate the performance of parallel machines. The **CG** kernel solves an unstructured sparse linear system by the conjugate gradient method. **EP** is an embarrassingly parallel kernel that generates pairs of Gaussian random deviates and tabulates the number of pairs in successive square annuli. **FT** is a fairly standard implementation of a 3D FFT PDE. **IS** is an integer sorting program; the keys are generated in an initial sequential portion. The **LU** solver application is a diagonal pipelining computation that results in a large number of small messages. **MG** is a simple 3D multigrid benchmark. The **SP** and **BT** applications each solve three sets of uncoupled systems of equations using a multi-partition scheme [BC88] which provides good load balance and uses coarse grained communication.

The benchmarks are MPI-based source-code implementations that are intended to run 'as is'. As we did not wish to alter the benchmarks, we decided to treat each program as a single 'superstep' for the purposes of evaluating the cost functions, that is, measurements (times and counters) were performed external to the benchmark execution using the SGI `Perfex` utility. This seemed a reasonable compromise as the NAS benchmarks have only a few barrier synchronizations, which are very fast on the SGI-PC, and, moreover, with the exception of LU, exchange small numbers of large messages, hence the overhead introduced by the MPI message-passing routines is rather limited. Finally, two of the benchmarks, BT and SP, required a square number of processors to run. In these cases, we used a nine-processor configuration of the machine but applied the functions derived for the eight-processor configuration, assuming that parameter values would not change significantly. Our assumption is confirmed by the very low errors obtained by F on these latter benchmarks.

Table 3 reports the errors incurred by the functions on each of the NPB benchmarks.

Here, the distinction between average and maximum errors does not make sense, since only one input size was used for each benchmark. For the DXBSP and the F3 functions we used the same estimates as the sorting applications for the number of L2 misses served by and writebacks to a particular memory bank. Again, it can be seen that the F function performs significantly better than the other functions, with almost perfect predictions for the EP, BT, and SP benchmarks, and errors of less than 75% in all cases. As with the sorting programs, we expect the optimistic versions of the BSP-like functions to perform much better than their pessimistic counterparts. However, even these are significantly worse than the F function, with errors up to 180% (errors are not shown for EBSP_{min} and $\text{DXBSP}_{\text{min}}$ because they were practically identical to the errors for QSM_{min}). Again, the reasonable level of accuracy attained by the optimistic versions of the BSP-like cost functions is to be attributed to the high locality and regularity exhibited by the benchmarks.

5.4 Extrapolating Performance

One of the advantages of the BSP-like functions over the counter-based function F, is that, to a large extent, the programmer can easily determine the input values for the function (e.g., H or M). However, as we have seen, these functions may not provide meaningful predictions as they all fail to account for hierarchy effects.

While the counter-based function exhibits excellent accuracy, it seems that one should actually run the program to obtain the required counts, which would annihilate its potential as a performance predictor. However, there are many cases where counter values can be guessed in advance with reasonable confidence, and then plugged in F to obtain accurate predictions. In fact, we claim that meaningful estimates for the counters can be derived by extrapolating values for large problem sizes from pilot runs of the program on small input sets.

To substantiate the above claim, we developed least-squares fits for each of the counters used in F for those supersteps in our three sorting algorithms that had significant communication. The input size n of the sorting instance was used as the independent variable. For each counter, we obtained the fits upon small input sizes ($n/p = 10^5 \cdot i$, for $1 \leq i \leq 5$), and then used the fits to forecast the counter values for large input sizes ($n/p = 10^5 \cdot i$, for $5 < i \leq 10$). These estimated counter values were then plugged in F to predict the execution times for the larger runs.

The results of this study are summarized in Table 4. Interestingly, but somewhat accidentally, in some cases the predictions obtained with the estimated counter values were actually slightly better than those obtained with the measured counter values (e.g., Column sort's superstep 1, Radix sort's superstep 1, and Sample sort's superstep 4). More importantly, however, it can be seen that in *all* cases, the level of accuracy of F using the extrapolated counter values was not significantly worse than what obtained with the actual counter values. These preliminary results indicate that a hardware counter-based function does indeed have potential as an *a priori* predictor of performance.

References

[ACF90] B. Alpern, L. Carter, and E. Feig. Uniform Memory Hierarchies. In *Proc. of the 31st IEEE FOCS*, pages 600–608, 1990.

[B+91] D. Bailey *et al.* The NAS parallel benchmarks. *Int. J. Supercomputer Appl.*, 5(3):63–73, 1991.

[B+95] D. Bailey *et al.* The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, http://www.nas.nasa.gov/npb/, 1995.

[BGMZ97] G.E. Blelloch, P.B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):943–958, 1997.

[BLM+98] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.

[BC88] J. Bruno and P. R. Cappello. Implementing the beam and warming method on the hypercube. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, jan 1988.

[GMR99] P.B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging-model for parallel computation? *Theory of Computing Systems*, 32, 1999.

[GLR+99] M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, and T. Tsantilas. Portable and efficient parallel computing using the BSP model. *IEEE Trans. Comput.*, c-48(7):670–689, 1999.

[JW98] B.H.H. Juurlink and H.A.G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 16(3):271-318, 1998.

[L85] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput.*, c-34(4):344–354, 1985.

[Sil95] Silicon Graphics Corporation. *SGI Power Challenge: User's Guide*, 1995.

[Sil97] Silicon Graphics Corporation. *Definition of MIPS R10000 Performance Counters*, 1997. `http://www.sgi.com/processors/r10k/performance.html`

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[WS88] Y. Won and S. Sahni. A balanced bin sort for hypercube multiprocessors. *Journal of Supercomputing*, 2:435–448, 1988.

[ZLTI96] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz Performance analysis using the MIPS R10000 performance counters. In *Proc. of Supercomputing'96*, Pittsburgh, PA 1996.

Code	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
p1.s1.b0	L1							
p8.s1.b0	L1	L1	L1	L1	L1	L1	L1	L1
p1.s8.b0	L2							
p8.s8.b0	L2	L2	L2	L2	L2	L2	L2	L2
p1.s32.b1/8	All							
p1.s64.b1/4	0,2,4,6							
p1.s128.b1/2	0,4							
p1.s256.b1	0							
p2.s32.b1/4	All	All						
p2.s64.b1/4	0,2,4,6	1,3,5,7						
p2.s64.b1/2	0,2,4,6	0,2,4,6						
p2.s128.b1/2	0,4	2,6						
p2.s128.b1	0,4	0,4						
p2.s256.b1	0	4						
p2.s256.b2	0	0						
p4.s32.b1/2	All	All	All	All				
p4.s64.b1/2	0,2,4,6	1,3,5,7	0,2,4,6	1,3,5,7				
p4.s128.b1/2	0,4	1,5	2,6	3,7				
p4.s64.b1	0,2,4,6	0,2,4,6	0,2,4,6	0,2,4,6				
p4.s128.b1	0,4	2,6	0,4	2,6				
p4.s256.b1	0	2	4	6				
p4.s128.b2	0,4	0,4	0,4	0,4				
p4.s256.b2	0	4	0	4				
p4.s256.b4	0	0	0	0				
p8.s32.b1	All	All	All	All	All	All	All	All
p8.s64.b1	0,2,4,6	1,3,5,7	0,2,4,6	1,3,5,7	0,2,4,6	1,3,5,7	0,2,4,6	1,3,5,7
p8.s128.b1	0,4	1,5	2,6	3,7	0,4	1,5	2,6	3,7
p8.s256.b1	0	1	2	3	4	5	6	7
p8.s64.b2	0,2,4,6	0,2,4,6	0,2,4,6	0,2,4,6	0,2,4,6	0,2,4,6	0,2,4,6	0,2,4,6
p8.s128.b2	0,4	2,6	0,4	2,6	0,4	2,6	0,4	2,6
p8.s256.b2	0	2	4	6	0	2	4	6
p8.s128.b4	0,4	0,4	0,4	0,4	0,4	0,4	0,4	0,4
p8.s256.b4	0	4	0	4	0	4	0	4
p8.s256.b8	0	0	0	0	0	0	0	0

Table 1: The basic suite of experiments. The three-field mnemonic code indicates, respectively te number of active processors, the stride and the resulting value of BKC. The table entry for a single processor indicates which of the 8 memory banks receive accesses from that processor.

Radix Sort Supersteps – Prediction Errors								
	SS1: Count Elts ($H = \frac{n}{p} + 64$)		SS2: Prefix Sums ($H = 128$)		SS3: Add Offsets ($H = 128$)		SS4: Move Elts ($H = \frac{2n}{p} + 64$)	
Function	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err
QSM_{min}	1.87	2.12	2.12	3.35	2.14	3.08	2.49	2.72
QSM_{max}	340.21	400.14	314.06	505.89	307.34	473.30	255.85	321.56
EBSP_{max}	223.56	320.48	187.95	298.79	180.15	295.88	180.25	302.11
$\text{DXBSP}_{\text{max}}$	219.83	258.55	202.93	326.88	198.59	305.83	165.32	207.78
F	1.25	1.41	1.39	1.86	1.38	1.84	1.15	1.39

Sample Sort Supersteps – Prediction Errors										
	SS1: Get Sample ($H = 200$)		SS2: Count Elts ($H = \frac{n}{p} + 2p$)		SS3: Prefix Sum ($H = 2p$)		SS4: Fill Bkts ($H = \frac{2n}{p} + 3p$)		SS5: Sort Bkts ($H = \frac{2n}{p}$)	
Function	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err
QSM_{min}	2.21	2.62	2.10	2.17	1.78	1.98	2.70	2.89	2.17	2.58
QSM_{max}	290.93	339.75	303.08	320.95	359.83	404.26	236.33	287.72	303.78	361.36
EBSP_{max}	155.25	196.86	230.71	252.25	182.16	195.76	191.96	247.31	275.44	327.08
$\text{DXBSP}_{\text{max}}$	187.98	219.53	195.84	207.38	232.50	261.21	152.70	185.91	196.29	233.49
F	1.25	1.44	1.06	1.15	1.22	1.30	1.08	1.11	1.15	1.26

Column Sort Supersteps – Prediction Errors										
	SS1: Init ($H = \frac{2n}{p}$)		SS2: Sort/Trans ($H = \frac{2n}{p}$)		SS3: Sort/RTrans ($H = \frac{2n}{p}$)		SS4: Sort ($H = \frac{2n}{p}$)		SS5:rs/Sort/ls ($H = \frac{2n}{p}$)	
Function	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err	Avg Err	Max Err
QSM_{min}	3.19	3.44	2.43	2.46	2.83	2.88	2.60	2.61	1.34	1.36
QSM_{max}	201.50	268.13	261.88	268.13	224.40	230.37	244.46	245.56	472.89	484.93
EBSP_{max}	148.73	205.23	259.82	264.49	223.23	228.11	245.32	247.10	273.33	280.03
$\text{DXBSP}_{\text{max}}$	130.20	173.25	169.22	173.25	144.99	148.85	157.96	158.67	305.56	313.34
F	1.02	1.06	1.86	2.05	1.73	1.88	1.87	2.09	1.15	1.16

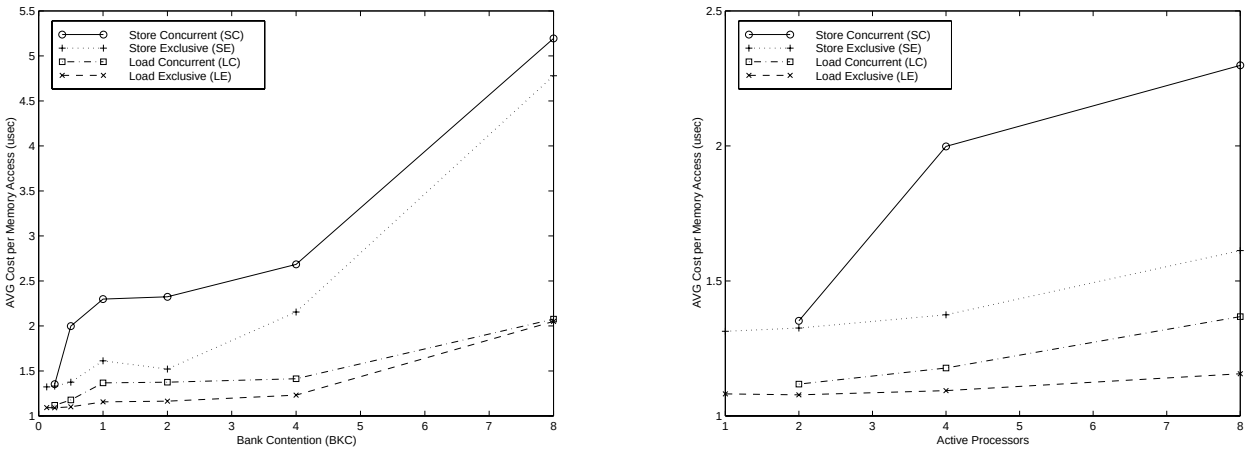
Table 2: Sorting algorithms: accuracy of the cost functions on individual supersteps. The errors (not shown) for EBSP_{min} and $\text{DXBSP}_{\text{min}}$ were practically identical to those for QSM_{min}.

NAS Parallel Benchmarks – Prediction Errors								
Function	**CG**	**EP**	**FT**	**IS**	**LU**	**MG**	**BT**	**SP**
QSM_{min}	2.46	2.42	2.05	1.57	2.15	1.57	2.77	2.13
QSM_{max}	258.31	262.53	309.40	404.81	295.01	403.48	229.68	298.69
EBSP_{max}	210.10	252.32	245.64	354.47	236.80	289.11	189.08	194.21
$\text{DXBSP}_{\text{max}}$	166.91	169.64	199.92	261.57	190.62	260.71	148.41	193.00
F	1.46	1.02	1.63	1.39	1.32	1.73	1.05	1.05

Table 3: NAS Parallel Benchmarks: accuracy of the cost functions on the individual benchmarks. The errors (not shown) for EBSP_{min} and $\text{DXBSP}_{\text{min}}$ were practically identical to those for QSM_{min}.

Sort	Superstep	Errors for F Measured Counts		Errors for F Estimated Counts	
		Average	Maximum	Average	Maximum
Radix	SS1: Count Elts	1.22	1.32	1.01	1.09
	SS4: Move Elts	1.11	1.16	1.13	1.16
Sample	SS2: Count Elts	1.05	1.09	1.20	1.21
	SS4: Fill Bkts	1.06	1.11	1.03	1.04
	SS5: Sort Bkts	1.13	1.17	1.24	1.26
Column	SS1: Init	1.12	2.49	1.17	1.72
	SS2: Sort/Trans	1.80	1.89	2.02	2.12
	SS3: Sort/RTrans	1.66	1.69	1.84	1.87
	SS4: Sort	1.78	1.83	2.05	2.06
	SS5: rs/Sort/ls	1.16	1.17	1.88	1.90

Table 4: Sorting algorithms: comparison of F's accuracy with measured vs estimated counters, over selected sorting supersteps and large input sizes. The errors shown above cover three test suites (each algorithm was run 16 times in each suite). Data from the first suite was somewhat noisy; if it is omitted, then the errors reported above would be lower (for both the measured and estimated counters). Figure 4 shows plots from the second suite.



(a) (b)

Figure 1: Access times plotted as functions of (a) bank and (b) bus contention. Access patterns selected to provide a range for both bank and bus contention; they were run in exclusive and concurrent modes, and for loads and stores, where applicable.

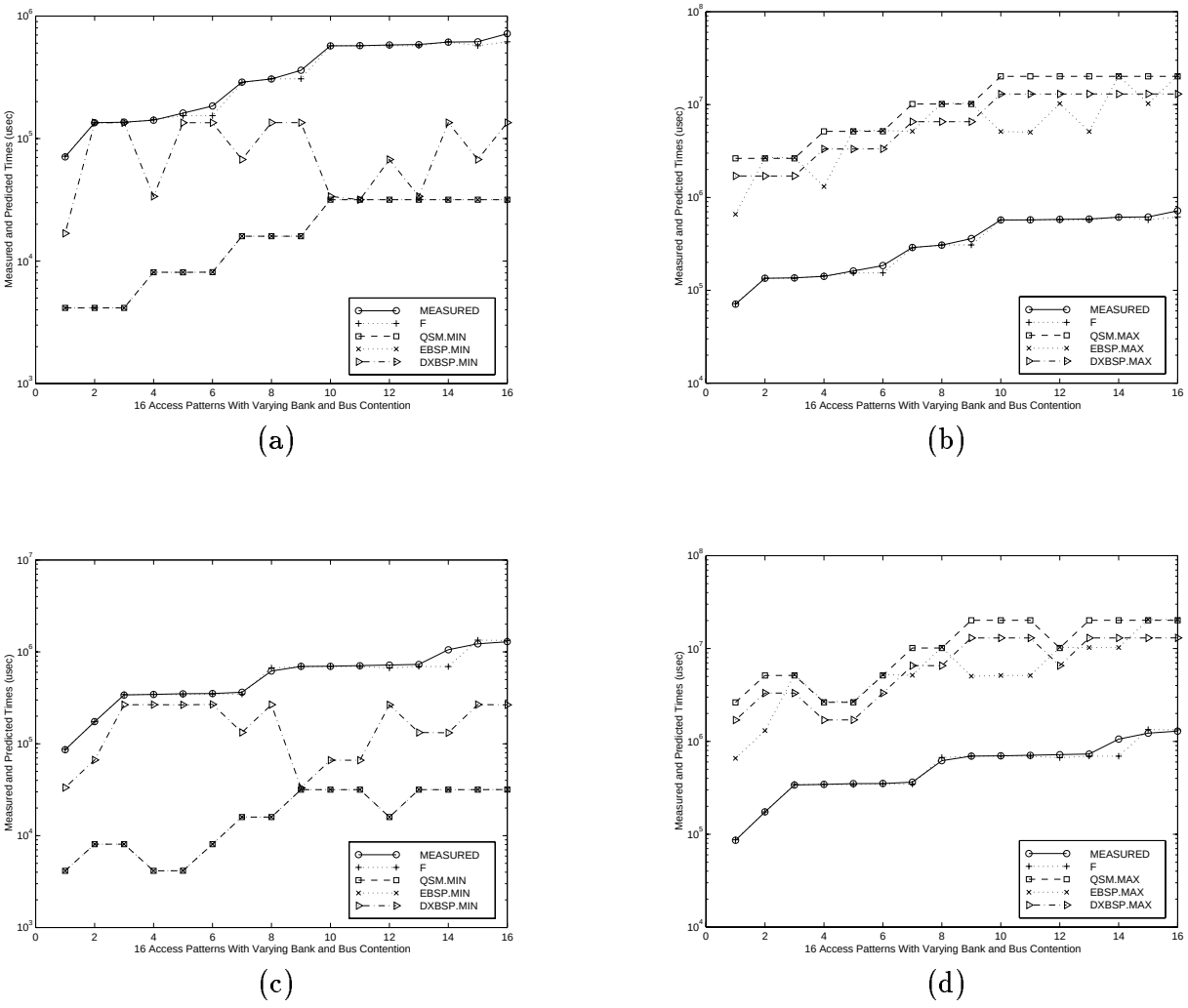


Figure 2: Plots of measured and predicted execution times for a set of microbenchmarks covering a range of bank and bus contention, and exclusive and concurrent accesses. The measured time and F are shown on all plots, (a) and (b) show, respectively, the optimistic and pessimistic versions of the BSP-like functions for Loads, and (c) and (d) show the same for Stores.

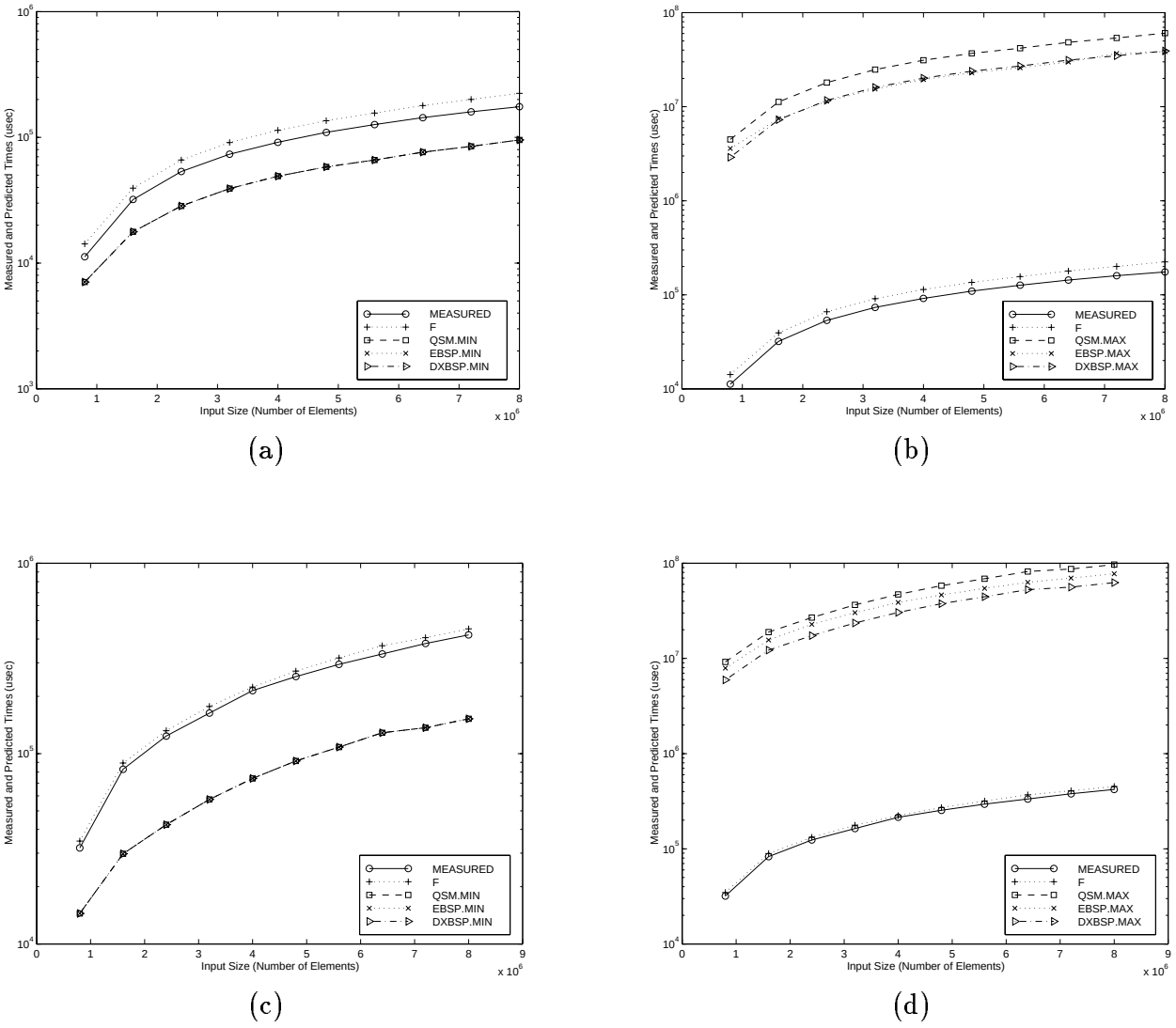


Figure 3: Plots of measured and predicted execution times for supersteps from Radix sort (SS1) and Sample Sort (SS4). The measured time and F are shown on all plots. (a) and (b) show, respectively, the optimistic and pessimistic versions of the BSP-like functions for Radix sort's SS1, and (c) and (d) show the same for Sample sort's SS4.

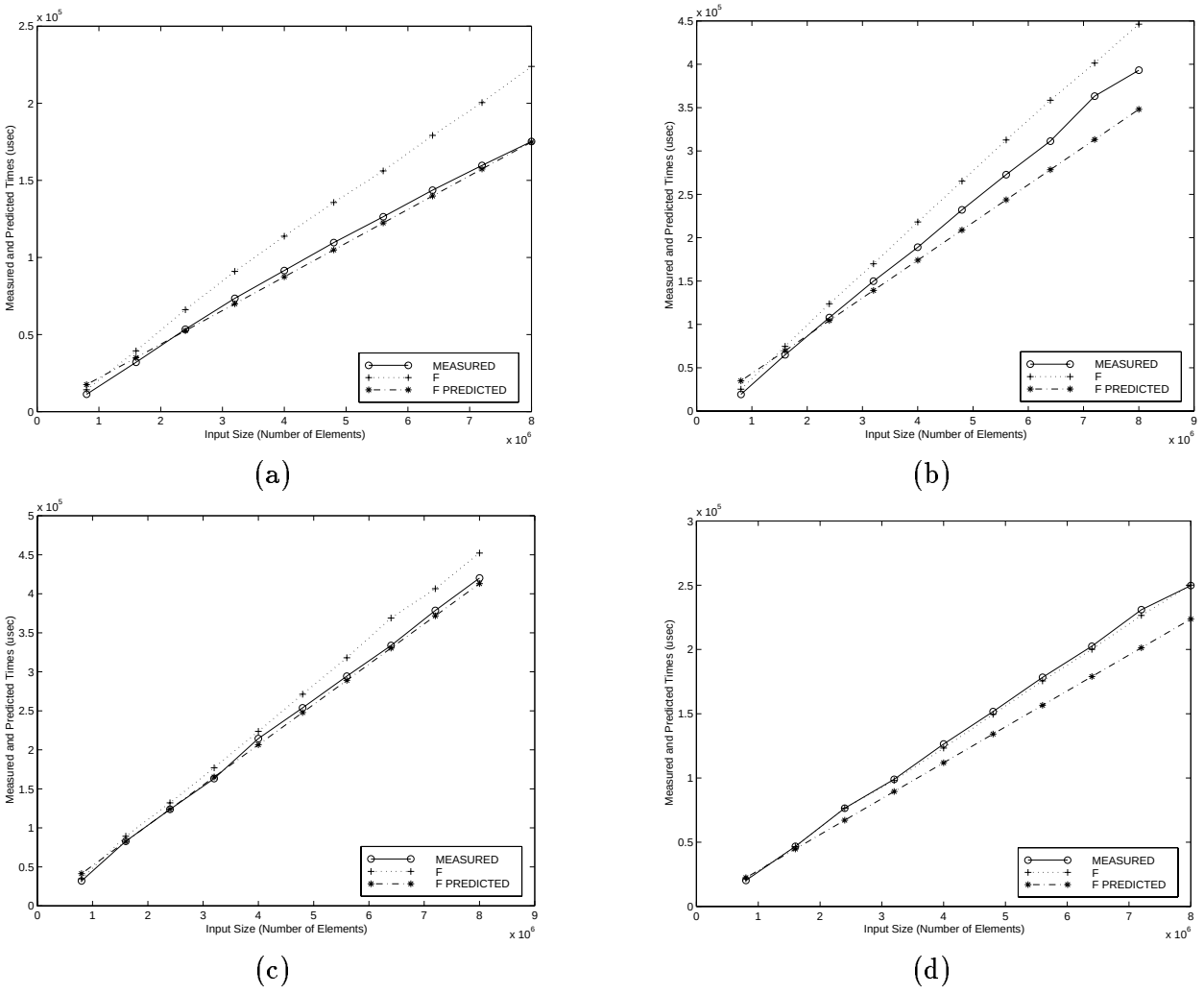


Figure 4: Plots of measured times, and times predicted by F, obtained using measured and estimated counters for (a) SS1 in Radix Sort, (b) SS4 in Radix Sort, (c) SS4 in Sample Sort, and (d) SS1 in Column Sort.