Scal-Tool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors[1]

Yan Solihin, **Vinh Lam**, and **Josep Torrellas**

Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
solihin,lam,torrellas@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

Abstract

Distributed Shared-Memory (DSM) multiprocessors provide an attractive combination of cost-effective commodity architecture and, thanks to the shared-memory abstraction, relative ease of programming. Unfortunately, it is well known that tuning applications for scalable performance in these machines is time-consuming. To address this problem, programmers use performance monitoring tools. However, these tools are often costly to run, especially if highly-processed information is desired. In addition, they usually cannot be used to experiment with hypothetical architecture organizations.

In this paper, we present *Scal-Tool*, a tool that isolates and quantifies scalability bottlenecks in parallel applications running on DSM machines. The scalability bottlenecks currently quantified include insufficient caching space, load imbalance, and synchronization. The tool is based on an empirical model that uses as inputs measurements from hardware event counters in the processor. A major advantage of the tool is that it is quite inexpensive to run: it only needs the event counter values for the application running with a few different processor counts and data set sizes. In addition, it provides ways to analyze variations of several machine parameters.

1 Introduction

Distributed Shared-Memory (DSM) multiprocessors are taking an increasing share of the market for medium- and large-scale parallel processing. These machines, built out of fast off-the-shelf microprocessors and sometimes even networks and nodes, exploit cost-effective commodity architectural designs. Examples of such machines are the SGI Origin 2000 [10], the HP-Convex Exemplar [1], the Sequent NUMAQ [11], the Data General NUMALiiNE [3], and the Sun WildFire [7].

In addition, it is argued that these machines deliver a programmable environment. The reason is the simplicity of the shared-memory abstraction presented to the programmer and the existence of automatic parallelizing compilers that, at least in theory, relieve the programmer from having to explicitly parallelize the codes.

In practice, it is well known that tuning applications for scalable performance in these machines is often a time-consuming effort. Indeed, the shared-memory abstraction hides widely-different costs, depending on whether the memory location accessed is cached or, if not, exists in local or in remote memory. Furthermore, state-of-the-art automatic parallelizing compilers still have considerable limitations.

As a result, computer manufacturers have developed a set of performance monitoring tools to assist the programmer in performance-tuning the applications. For example, SGI provides, among other tools: *perfex* [18], which is a set of hardware event counters in the processor that can record up to 32 hardware events; *speedshop* [19], which profiles the cycles spent in each routine of the application or library; and *ssusage* [20], which measures the maximum number of pages that an application has at any time in memory. With these tools, the programmer can find bottlenecks in the code and tune it for better scalability.

Unfortunately, these tools are often costly to run, especially if we want to obtain highly-processed information. For example, suppose that we want to measure the total time that a parallel program spends synchronizing or spinning due to load imbalance, for different processor counts (1, 2, 4, ... $2^{(n-1)}$). For each processor count, we run *time* and the more intrusive *speedshop*. We use *time* to measure the execution time, while we use *speedshop* to measure the fraction of the cycles spent in the synchronization and spinning routines. Table 1 lists the number of runs that must be performed, the total number of processors required, and the number of output files that must be analyzed. Although the number of files could be reduced by generating a single file in every *speedshop* run, all these activities clearly involve significant effort and

[1]This work was supported in part by the National Science Foundation under grants NCSA-PACI ACI 96-19019COOP, NSF Young Investigator Award MIP-9457436, ASC-9612099, and MIP-9619351, gifts from Intel and IBM, and NCSA machine time under grant AST910367N.

Parameter Measured: (*Tool*)	Num. Runs	Total Num. Processors	Num. Files
Execution Time: (*time*)	n	$2^n - 1$	n
Fraction of Cycles in Synch. and Spinning: (*speedshop*)	n	$2^n - 1$	$2^n - 1$
Total with Existing Tools	2n	$2^{n+1} - 2$	$2^n + n - 1$
Total with *Scal-Tool*	2n-1	$2^n + n - 2$	2n-1

Table 1: Resources needed by the existing performance tools and by *Scal-Tool* for our example.

resource usage.

In addition, most performance monitoring tools can only measure statistics on the actual machine. As a result, they can rarely be used to experiment with hypothetical architecture organizations. For example, it is usually hard to estimate the effect of doubling the L2 cache size on application performance.

In this paper, we introduce a new tool to isolate and quantify scalability bottlenecks in parallel applications. The tool, called *Scal-Tool*, does it in a relatively inexpensive and integrated manner, and allows room to experiment with hypothetical architecture organizations. *Scal-Tool* is based on an empirical model that uses Cycles Per Instruction (CPI) breakdown equations. It uses as inputs the measurements from hardware event counters in the processor. It isolates and quantitatively estimates the cycle count impact of different scalability bottlenecks, such as insufficient caching space, load imbalance, and synchronization.

An important characteristic of *Scal-Tool* is that it is relatively inexpensive to run. For example, as we will see in Section 2, to obtain the cost of synchronization and load imbalance as in the previous example, we need to run the program with the base data set size for each processor count (1, 2, 4, ... $2^{(n-1)}$) once. In addition, we need to run the program on a uniprocessor with n different data set sizes, one of which is the base one. In all cases, we generate a single file. Overall, *Scal-Tool* requires a total of $2n-1$ runs, 2^n+n-2 processors, and generates $2n-1$ files (Table 1). Compared to the requirements of the existing tools, for runs up to 32 processors ($n = 6$), *Scal-Tool* needs only about 50% of the processors and generates fewer files. The result is significant savings in processor use and file handling time.

In addition, *Scal-Tool* provides some ways to experiment with different machine parameters and evaluate their impact on application performance. Possible machine parameters include L2 cache size, memory hierarchy speed (L2 cache, main memory, network interconnection), and synchronization support.

In the rest of the paper, we describe the empirical scalability model at the heart of *Scal-Tool* (Section 2), discuss the SGI Origin 2000 platform that we use to evaluate *Scal-Tool* (Section 3), evaluate and validate *Scal-Tool* with real applications (Section 4), and review related work (Section 5).

2 The Scalability Model

To describe the model, we present the overall picture first, and then discuss how we estimate the different parameters.

2.1 Overall Picture

We follow the work of Lubeck *et al* [12, 24] to compute the overall CPI of an application. We call cpi_0 the average CPI of all the instructions in the program excluding any cache miss time. This term might be called the compute CPI. We then consider two types of cache-missing instructions: (i) those that miss in the L1 cache and hit in the L2 cache, and (ii) those that miss in the L2 cache and, therefore, access main memory. These instructions occur with a frequency of $h2$ and hm, respectively. They have an average CPI beyond cpi_0 of $t2$ and tm, respectively. Consequently, for a given application, we have:

$$cpi = cpi_0 + h2 \cdot t2 + hm \cdot tm \qquad (1)$$

cpi, $h2$, and hm can be measured directly with hardware event counters in the processor chip. Indeed, cpi is the ratio between the program execution time and the number of committed instructions. The latter, also called graduated instructions, exclude instructions from wrongly-speculated paths. hm is the ratio between L2 cache misses and committed instructions. Finally, $h2$ is the difference between L1 and L2 cache misses divided by the committed instructions. In contrast, cpi_0, $t2$, and tm have to be estimated. In Sections 2.2 and 2.3, we show how this is done. After these parameters are estimated, we can use Equation 1 to isolate and quantify the various scalability bottlenecks (Section 2.4).

There are several bottlenecks that affect the speedup of applications as we increase the number of processors. Two major bottlenecks are insufficient caching space to

Bottleneck		Effects
Insufficient Caching Space		Conflict Misses
Multiprocessor Factors	Synchronization	Coherence Misses + Extra Instructions
	Load Imbalance	Extra Instructions
	True Sharing	Coherence Misses
	False Sharing	Coherence Misses

Table 2: Bottlenecks that affect the scalability of applications and their effects.

hold the working set of the application, and multiprocessor factors. Table 2 analyzes these bottlenecks.

As shown in the table, the result of insufficient caching space is conflict misses. In this paper, we call conflict misses the combination of what is often referred to as capacity and conflict misses [8]. These conflict misses slow down the application, especially for low processor counts. The multiprocessor factors include synchronization, load imbalance, and true and false sharing [22]. The effects of synchronization include the coherence misses induced by cache line invalidations and the extra instructions involved in the synchronization. The effects of load imbalance include the extra instructions induced by idle thread spinning. Finally, true and false sharing cause coherence misses.

Our model aims at isolating and quantifying the effect of these bottlenecks inexpensively. The result of the model is a set of curves like those in Figure 1 for each application. The curves show the execution time of the application as we increase the processor count. The top most curve (*Base*) is obtained by measuring the execution time on the real machine. Our model generates the second curve by removing the estimated effect of the insufficient caching space. This bottleneck has the highest impact for low processor counts and tends to become negligible for higher processor counts. Finally, our model produces the bottom most curve by further removing the estimated effects of one or several of the multiprocessor factors. These effects are zero for 1-processor runs and increase as the processor count increases. In general, the different effects can be removed in any order.

Overall, by generating these types of charts, our model gives insight to the programmer as to what bottlenecks affect the performance of the application the most. The programmer can then try to remove the bottlenecks. Note that these plots can be obtained for the overall application or for a segment of the application that is considered particularly important.

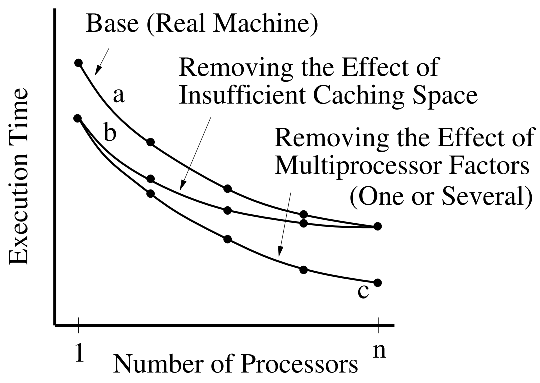

Figure 1: Execution time of an application under real and estimated conditions.

2.2 Estimating cpi_0

cpi_0 is the average CPI of all the instructions in the program excluding any cache miss time. Our model neglects instruction misses. For a given program, cpi_0 may depend on the data set size and the processor count. However, following the work by Lubeck *et al* [12, 24], we assume that cpi_0 is largely constant and compute an average value. Note, however, that our work is quite different from that of Lubeck *et al.* For one thing, they model a uniprocessor, while we model parallel systems.

To estimate cpi_0, we cannot use the ideal CPI of the program where cache accesses always hit as estimated by SGI's *speedshop* [19] because the estimated value is often highly erroneous. Lubeck *et al* [12, 24] estimate the value by measuring the overall CPI of the program running on one processor with a data set size that fits in the L1 data cache. The assumption is that misses are then negligible. Unfortunately, this estimate is biased because the execution necessarily includes some compulsory misses.

In our model, we introduce an unbiased estimator of cpi_0: we adjust Lubeck's method by removing the effect of compulsory misses. First, we make an initial estimation of cpi_0 as in Lubeck's. Then, we use cpi_0 to estimate $t2$ and tm as described in Section 2.3. Finally, cpi_0 is adjusted to exclude the $t2$ and tm cycles induced by the compulsory misses at the point of cpi_0. Specifically, if the initial estimation of cpi_0 is denoted by $\widehat{cpi_0^{(1)}}$, and the estimated $t2$ and tm are $\hat{t2}$ and $\hat{tm}$, the adjusted cpi_0, which is denoted by $\widehat{cpi_0^{(2)}}$, is computed with:

$$\widehat{cpi_0^{(2)}} = \widehat{cpi_0^{(1)}} - h2 \cdot \hat{t2} - hm \cdot \hat{tm} \qquad (2)$$

In the equation, $h2$ and hm are measured with the hardware event counters in the processor at the point of $\widehat{cpi_0^{(1)}}$. The resulting $\widehat{cpi_0^{(2)}}$ is an unbiased estimator of cpi_0.

2.3 Estimating $t2$ and tm

For a given program, $t2$ and tm may vary with the data set size and the processor count. However, to simplify, we assume that $t2$ is largely constant and that tm changes only with the processor count. tm changes with the processor count because, with more processors, the physical dimensions of the machine are larger and, therefore, accesses to main memory take longer.

In the previous subsection, we indicated that we need $t2$ and tm (the latter for one processor), to compute $\widehat{cpi_0^{(2)}}$. To measure $t2$ and tm, the program is run on one processor with varying data set sizes. We apply Equation 1 repeatedly:

$$\begin{aligned} cpi^1 &= \widehat{cpi_0^{(1)}} + h2^1 \cdot t2 + hm^1 \cdot tm \\ cpi^2 &= \widehat{cpi_0^{(1)}} + h2^2 \cdot t2 + hm^2 \cdot tm \\ &\ldots \end{aligned} \qquad (3)$$

Ideally, since there are only two unknowns ($t2$, tm), we only need two sets of triplets (cpi, $h2$, hm) measured with the hardware event counters in the processor. However, considering the random nature of the numbers measured, the more triplets measured, the better we can estimate the two unknowns with the least square error method [16]. For the different triplets to be significant, however, they must correspond to a variety of data set sizes. Consequently, we measure triplets for about 3-4 data set sizes. Then, using the least square error method, we compute $t2$ and tm.

It should be noted that, for this method to give accurate results, the values of $t2$ and tm should vary as little as possible across the different data set sizes used to generate triplets. In practice, we find that tm varies noticeably depending on whether or not the data set size fits in the L2 cache. Consequently, we use only data set sizes that overflow the L2 cache when we generate the triplets.

Overall, the method described in this subsection and the previous one delivers cpi_0, $t2$ and, for one processor, tm. We can now estimate tm for different processor counts by running the program for the base data set size s_0 for different processor counts. For each processor count, we use Equation 1 to estimate tm. In the equation, cpi_0 and $t2$ are known, while cpi, $h2$, and hm are measured with the hardware event counters.

2.4 Isolating the Effects of Bottlenecks

Each of the curves in Figure 1 can be computed from the CPI of the application as shown in Figure 2. In general, the CPI of an application $cpi(s, n)$ is a function of the data set size s and the processor count n. As shown in Figure 2, curve a is simply $cpi(s_0, n)$ times the number of instructions $inst$. Both $cpi(s_0, n)$ and $inst$ are measured with the hardware event counters.

Curve b is $cpi_\infty(s_0, n)$ times the number of instructions $inst$. $cpi_\infty(s_0, n)$ is the CPI of the application after removing any caching space limitation. It needs to be estimated. In our model, having no cache space limitation means to have an infinite L2 cache. The size of the L1 cache is unchanged because the L1 cache is often on chip, and cache upgrades usually involve increasing the size of the L2 cache only.

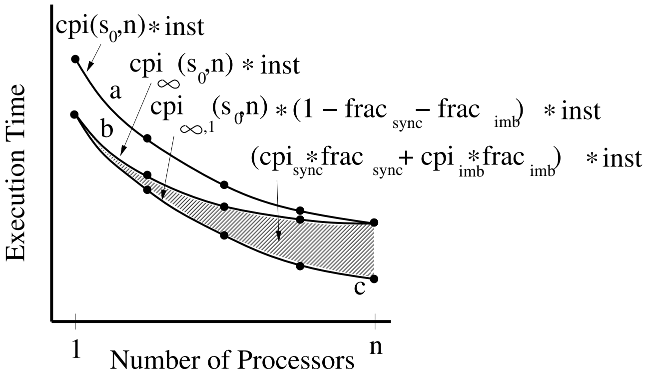

Figure 2: Breakdown of the execution time of an application under real and estimated conditions as a function of the CPI. The base data set size we call s_0.

To obtain curve c, we remove both the cache space limitations and all the multiprocessor factors. We need to estimate the new CPI and the new number of instructions. The new CPI is $cpi_{\infty,1}(s_0, n)$, which is the CPI after removing both cache space limitations and all multiprocessor effects. The number of instructions is different than before. We must subtract the extra instructions induced by the multiprocessor factors. These instructions are $(frac_{sync} + frac_{imb}) \cdot inst$, where $frac_{sync}$ and $frac_{imb}$ are the fraction of instructions induced by synchronization and load imbalance spinning respectively.

The effect of the multiprocessor factors is shown as the

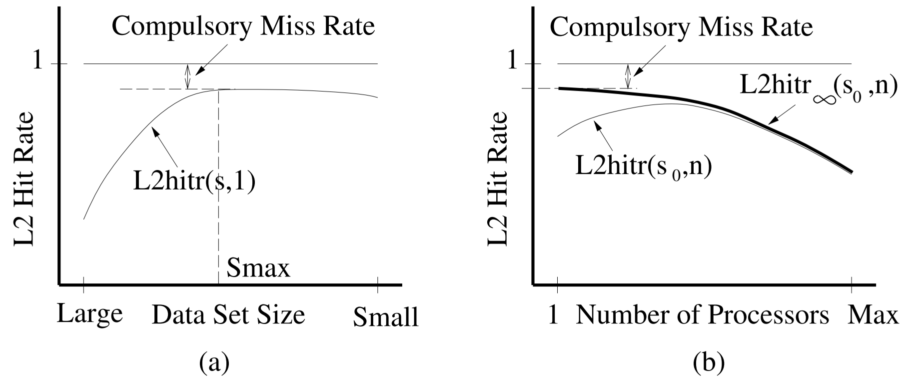

Figure 3: Removing the effects of insufficient caching space.

shaded area in Figure 2. In the scientific applications that we analyze (Section 4), the effects of true and false sharing are largely negligible. Consequently, although our model could be extended to take them into account, we neglect them. As a result, if cpi_{sync} and cpi_{imb} are the CPIs of the instructions involved in synchronization and load imbalance spinning respectively, the multiprocessor effects are:

$$(cpi_{sync} \cdot frac_{sync} + cpi_{imb} \cdot frac_{imb}) \cdot inst \qquad (4)$$

Overall, to plot Figure 2 and separate synchronization from load imbalance effects, we need to estimate $cpi_{\infty}(s_0, n)$, $cpi_{\infty,1}(s_0, n)$, $frac_{sync}$, $frac_{imb}$, cpi_{sync}, and cpi_{imb}. The following sections show how to do it.

To set the stage for estimating these parameters, let us write $cpi(s, n)$ as a function of $L2hitr(s, n)$, the *local* hit rate of the L2 cache. Consider again the basic equation:

$$cpi(s, n) = cpi_0 + h2(s, n) \cdot t2 + hm(s, n) \cdot tm(n) \qquad (5)$$

Recall that we assume that cpi_0 and $t2$ are constant and that $tm(n)$ only depends on the processor count n. It is trivial to rewrite $h2(s, n)$ and $hm(s, n)$ as a function of the local hit rates of the two caches $L2hitr(s, n)$ and $L1hitr(s, n)$, and of the fraction of memory instructions $m(s, n) = (loads + stores)/instructions$. Note that $loads$, $stores$, and the hit rates include the contribution of some memory accesses that will never be committed. While we cannot eliminate such a contribution, we expect that it will have a largely negligible effect on our results.

$$\begin{aligned} h2(s, n) \quad &= \quad (1 - L1hitr(s, n)) \cdot m(s, n) \\ &\quad \cdot L2hitr(s, n) \end{aligned} \qquad (6)$$

$$\begin{aligned} hm(s, n) \quad &= \quad (1 - L1hitr(s, n)) \cdot m(s, n) \\ &\quad \cdot (1 - L2hitr(s, n)) \end{aligned} \qquad (7)$$

Replacing expressions, we have that:

$$\begin{aligned} cpi(s, n) \quad &= \quad cpi_0 + (1 - L1hitr(s, n)) \cdot m(s, n) \\ &\quad \cdot (tm(n) + (t2 - tm(n)) \cdot L2hitr(s, n)) \end{aligned} \qquad (8)$$

2.4.1 Removing the Effects of Insufficient Caching Space

To estimate curve b in Figure 2, we need to estimate $cpi_{\infty}(s_0, n)$. To do so, Equation 8 shows that we only need to estimate $L2hitr_{\infty}(s_0, n)$. The reason is that the value of $tm(n)$ has been computed in Section 2.3. In addition, $L1hitr(s_0, n)$ and $m(s_0, n)$ should change negligibly with an infinite increase in the L2 cache size. Consequently, we use their values measured from the real machine.

To estimate $L2hitr_{\infty}(s_0, n)$, we need to isolate the compulsory and coherence miss rates of the application and add them up. The result is the desired miss rate. Since we are assuming an infinite L2 cache, there are no conflict misses. Such conflict misses are the difference between $L2hitr_{\infty}(s_0, n)$ and the measured $L2hitr(s_0, n)$. In the following, we show how to estimate the compulsory and coherence miss rates.

To estimate the compulsory miss rate, we measure the L2 hit rate for the application running on a single processor as we reduce the data set size. This is shown in Figure 3-(a) as the curve labeled $L2hitr(s, 1)$. For large data set sizes, on the left side, the hit rate is low because there are conflict misses. As the data set size decreases to the right, the hit rate increases. At a certain s_{max}, the maximum point is reached, when only the compulsory miss rate remains. For smaller data set sizes, the hit rate decreases slightly again because the few instruction misses that exist have a relatively bigger weight. From this chart, we remember the compulsory miss rate.

To estimate the coherence miss rate, we compare the measured multiprocessor hit rate ($L2hitr(s_0,n)$) to the hit rate of uniprocessor runs. The difference is the estimated coherence miss rate. These uniprocessor runs, however, are performed on an *nth* fraction of the data set size ($L2hitr(s_0/n,1)$). This adjustment is done to compensate for the larger L2 caching space that an n-processor run has. The assumption in using this adjustment is as follows: the non-coherence L2 miss rate recorded by one processor in an n-processor run is the same as the same parameter in a one-processor run of one-*nth* the data set size. Subtracting the hit rate of the n-processor run with s_0 data set size from the hit rate of the uniprocessor run with s_0/n data set size, we obtain the estimate of the coherence miss rate for the n-processor run. We call this function $Coh(s_0,n)$. The need to obtain these values explains why we need the single-processor runs referred to in Section 1. If an application does not allow the slicing of the data set to the right size, we interpolate between the results of two acceptable data set sizes.

Finally, $L2hitr_\infty(s_0,n)$ is simply 1 minus the compulsory miss rate and minus $Coh(s_0,n)$. The resulting curve as a function of the processor count n is shown in Figure 3-(b). We can compare the curve to the measured multiprocessor hit rate ($L2hitr(s_0,n)$, also shown in the figure). For the uniprocessor run, $L2hitr_\infty(s_0,n)$ is exactly 1 minus the compulsory miss rate. At that point, $L2hitr_\infty(s_0,n)$ is higher than $L2hitr(s_0,n)$ because the latter curve includes conflict misses. For runs with a high processor count, however, $L2hitr_\infty(s_0,n)$ decreases because more coherence misses appear. In the limit, the $L2hitr_\infty(s_0,n)$ and $L2hitr(s_0,n)$ curves converge because, with many processors, the real machine has such a large caching space that the conflict miss rate is negligible. Overall, the resulting $L2hitr_\infty(s_0,n)$ curve is used in Equation 8 to compute $cpi_\infty(s_0,n)$ for Figure 2.

2.4.2 Removing the Effects of Multiprocessor Factors

To estimate curve c in Figure 2, we need to estimate $cpi_{\infty,1}(s_0,n)$, $frac_{sync}$, and $frac_{imb}$. To estimate $cpi_{\infty,1}(s_0,n)$, Equation 8 shows that we need to estimate $L1hitr_{\infty,1}(s_0,n)$, $m_{\infty,1}(s_0,n)$, and $L2hitr_{\infty,1}(s_0,n)$.

To estimate $L1hitr_{\infty,1}(s_0,n)$ and $m_{\infty,1}(s_0,n)$, we make the same assumption as in Section 2.4.1, and use the values measured for single-processor runs with adjusted data set sizes, namely $L1hitr(s_0/n,1)$ and $m(s_0/n,1)$. The assumption implies that the non-coherence component of both L1 miss rates and fraction of memory instructions recorded by one processor in an n-processor run is the same as the same parameter in a uniprocessor run of one-*nth* the data set size.

To estimate $L2hitr_{\infty,1}(s_0,n)$, we need to remove the effect of the coherence activity from $L2hitr_\infty(s_0,n)$. The result is simply that $L2hitr_{\infty,1}(s_0,n)$ is 1 minus the compulsory miss rate of Figure 3-(a).

Typically, $cpi_{\infty,1}(s_0,n)$ increases with increasing processor count n. One major reason is because $cpi_{\infty,1}(s_0,n)$ depends on $tm(n)$ (Equation 8), which itself increases with n. Intuitively, the larger machine size induces a longer latency on each of the compulsory misses. Figure 4 shows a typical variation of $cpi_{\infty,1}(s_0,n)$ with the processor count.

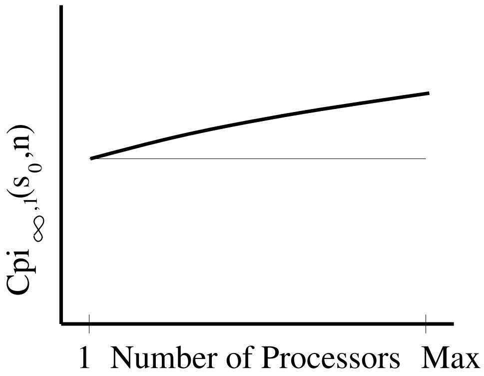

Figure 4: Removing the effects of insufficient caching space and multiprocessor factors.

To estimate $frac_{sync}$ and $frac_{imb}$, we use the following formula, which results from Figure 2:

$$\begin{aligned} cpi_\infty(s_0,n) &= cpi_{\infty,1}(s_0,n)\cdot(1-frac_{sync}-frac_{imb}) \\ &\quad +cpi_{sync}\cdot frac_{sync} \\ &\quad +cpi_{imb}\cdot frac_{imb} \end{aligned} \tag{9}$$

We estimate cpi_{sync} and cpi_{imb} by running small, synthetic kernels that continuously synchronize and spin in an idle loop, respectively. The hardware event counters tell us the CPI. Note that the kernel for synchronization should not include spinning; it is simply a loop where processors come in and out of barriers. cpi_{sync} is found to be a function of n.

Of the two remaining unknowns in Equation 9, we can estimate $frac_{sync}$ with one of two methods. One method is to instrument the application to count, at run time, the number of barriers that the processors go through. Both explicit and implicit barriers are counted. Since we know the instruction cost for each barrier, we can compute $frac_{sync}$. If the application has locks, we need to separately compute the cpi_{sync} of a kernel of locks and count at run-time the number of locks executed.

The second method involves trying to estimate first the overall spin-free cost of synchronization. To do so, we use an approach that is specific to the platform that we use, namely the SGI Origin 2000 [10] (Section 3). We use a hardware event counter that is incremented when the processor stores on a location that it already has in state shared [25]. Since the Origin 2000 uses the Illinois cache coherence protocol [14], such operations largely imply sharing transactions. Furthermore, since we use applications with little true or false sharing, this counter is largely incremented by synchronization operations. Let us call the value of this counter for an application cnt_{sync}.

In the Origin2000, synchronization is implemented by the fetchop facility for atomic operations [17]. While the information available to us regarding the actual implementation of this primitive is sketchy, it appears that every acquire to a synchronization variable involves one full memory access. We estimate the latency of such an access by using a kernel of synchronizations, and proceeding like we did to calculate tm. Let us call the latency t_{sync}. Consequently, the cost of synchronization can be estimated as:

$$cost_{sync} = cnt_{sync} \cdot (cpi_0 + t_{sync}) \qquad (10)$$

Since $cost_{sync}$ is also $cpi_{sync} \cdot frac_{sync}$, we can obtain an estimate of $frac_{sync}$. This is the method that we use in Section 4.

Finally, we can compute the only unknown from Equation 9, namely $frac_{imb}$. Overall, we can now completely plot curve c of Figure 2. In addition, we can also divide the shaded area in Figure 2 into the contribution of synchronization and that of load imbalance.

2.5 Overall Resource Requirements

From the previous discussion, we conclude that, given an application, we can obtain the curves in Figure 2, and separate the shaded region in the figure into the synchronization and load imbalance components, quite inexpensively. We need to run the application with the base data set size s_0 for each processor count (1, 2, 4, ... $2^{(n-1)}$) once (Table 3). In addition, we run the application on a uniprocessor with fractional data set sizes ($s_0/2$, $s_0/4$, ... $s_0/2^{(n-1)}$) once. Among these latter runs, those that have data set sizes that overflow the L2 cache can also be used as different data points to estimate $t2$ and tm (Section 2.3). In each run, we read the hardware event counters and generate a single output file. We have, therefore, validated our claims in the example of Table 1.

		Processor Count				
		1	2	4	...	$2^{(n-1)}$
Data Set Size	s_0	x	x	x	x	x
	$s_0/2$	x				
	$s_0/4$	x				
	...	x				
	$s_0/2^{(n-1)}$	x				

Table 3: Runs needed to gather the empirical data necessary for *Scal-Tool*. s_0 is the base data set size.

2.6 Experimenting with Different Parameters

Before we finish the description of *Scal-Tool*, we show that we can also use it to evaluate the performance impact of changing some machine parameters. The idea is to modify the values of the parameters in the model and use the model equations to infer the rough performance impact on the application. The application does not need to be re-run.

For example, we can estimate the impact of faster or slower L2 caches, interconnection network, and synchronization support by changing the latency parameters $t2$, tm, and t_{sync}, respectively. Furthermore, we can estimate the impact of changing the processor issue width by changing cpi_0.

In another example, we can also roughly estimate the impact on the L2 miss rate of increasing the L2 cache sizes of the multiprocessor by a factor of k. For this, we conceptually divide the L2 miss rate into two components: coherence component and uniprocessor component. We assume that, to a first approximation, the coherence component depends only on the processor count n and is not affected by the L2 cache size. Using assumptions described before, such coherence miss rate is:

$$Coh(s_0, n) = L2hitr(s_0/n, 1) - L2hitr(s_0, n) \qquad (11)$$

We then assume that the uniprocessor component changes, to a first approximation, as follows: increasing the L2 cache size by a factor of k is like reducing the data set size of the application by a factor of k. In that case, the uniprocessor component of the miss rate is $1 - L2hitr(s_0/k, 1)$. Adding up the two miss rate components produces a rough estimate of the overall miss rate. Note that we do not re-run the application.

Finally, as another example, we can also estimate the impact of using a new synchronization primitive. We can design a kernel that simulates the new primitive and measure the resulting cpi_{sync} of the primitive. The new cpi_{sync} is then used in the model equations. In this case, however, it is harder to predict the actual performance change because synchronization performance may impact load imbalance.

3 Platform & Applications

To evaluate *Scal-Tool*, we use a SGI Origin 2000 [10]. The Origin 2000 is a cache-coherent DSM machine based on 250-MHz MIPS R10000 processors connected in a bristled hypercube interconnection. Each processor has a 32-Kbyte L1 data cache, a 32-Kbyte L1 instruction cache, and a 4-Mbyte unified L2 cache. Each processor has two hardware event counters that can record up to 32 events [18, 25]. Some of the events measured include the number of cache misses, cycles, and instructions. The machine is fully cache coherent in hardware, supported by a directory-based scheme using bit vectors.

The models of parallelism supported by our model are MP and PCF. MP extracts parallelism from loops using the DOACROSS primitive. The default policy is to use block scheduling to schedule iterations and first-touch to allocate pages in memory. PCF (Parallel Computing Forum) uses a more powerful set of primitives and a more general model of parallelism. PCF uses constructs like critical sections, single-processor execution, non-loop parallelism, and explicit barrier directives. In all cases, we compile the applications using -O3 before running them on the Origin 2000.

For our experiments, we use three applications: T3dheat from Los Alamos National Laboratory, and Hydro2d and Swim from the SPECFP95 suite [21]. Table 4 shows that they have various degrees of speedup and load balance, and use different models of parallelism.

4 Evaluation

To evaluate *Scal-Tool*, we use it to analyze the scalability bottlenecks of the applications. We also validate the tool. We consider each application in turn.

4.1 T3dheat

The speedup curve for T3dheat is depicted in Figure 5. It shows that the Origin 2000 delivers good speedups up to 16 processors. However, after that, the curve saturates. To understand the speedups, we use *Scal-Tool* on T3dheat (Figure 6). Figure 6 is organized as Figure 1 except that the curves accumulate the cycles from *all the processors* – not the execution time. The figure has four curves. *Base* is the cycles measured in the Origin 2000. The other curves are obtained by subtracting from *Base* the different bottlenecks discussed in Section 2.4. Specifically, *L2Lim*, *Sync*, and *Imb* are the estimated effects induced by the insufficient caching space, synchronization, and load imbalance respectively. For example, *Base-L2Lim* is the cycles measured in the application minus the effect of insufficient caching space. *MP* is the total multiprocessor cost (*Sync*+*Imb*).

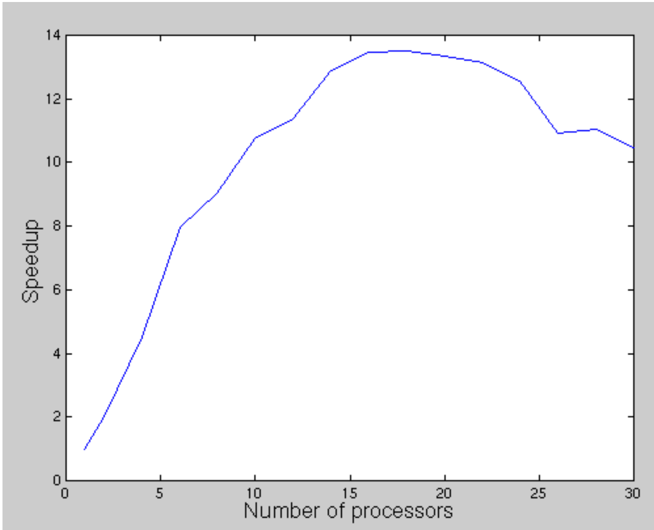

Figure 5: Speedups for T3dheat.

The *Base-L2Lim* curve, which is hard to distinguish in the figure, has the following shape. For 4 processors or less, it overlaps with all the other curves except *Base*. For 8 processors or more, it overlaps with *Base*. Consequently, the figure shows that, for 1 processor, the overhead induced by the limited L2 cache, namely the conflict misses, is significant: it is responsible for nearly doubling the execution time. This effect gradually decreases with the processor count and becomes zero at 8 processors. However, at that point, multiprocessor overheads start increasing. They keep on increasing until they are responsible for about 75% of the cycles for 30 processors. The figure also shows that most of the multiprocessor overhead comes from synchronization.

The data clearly shows that this is not a scalable application. The excellent speedup obtained up to 16 processors is simply because the application does not fit in the caches for low processor counts. As a result, additional processors, which provide more caching space, help run the application much faster, delivering linear speedup. Beyond 8 processors, there is no cache space limitation, but synchronization costs start to increase fast, saturating the speedup.

To validate the estimated *L2Lim* effect to some extent, we use SGI's *ssusage* tool to measure T3dheat's data set size. It is found to be 40 Mbytes. Given that the L2 cache sizes are 4 Mbytes, if the per-processor working sets

Application	Source	What It Does	Scalability & Load Balance	Data Set Size (Mbytes)	Parameters	Model of Parallelism
T3dheat	Los Alamos National Laboratory	PDE solver using conjug. gradient	Excellent scalability up to 16, poor beyond 16. Good load balance	40	imax=jmax= =kmax=50 5 iters	PCF directives with explicit barriers
Hydro2d	SPECFP95	Shallow water simulation	Modest scalability (9 at 32 processors). Large serial sections	10.3	N=40 istep=5	MP directives with DOACROSS
Swim	SPECFP95	Navier Stokes equations	Good scalability (24 at 32 processors). Good load balance	16.2	M=N=512 100 iters	MP directives with DOACROSS

Table 4: Characteristics of the applications analyzed.

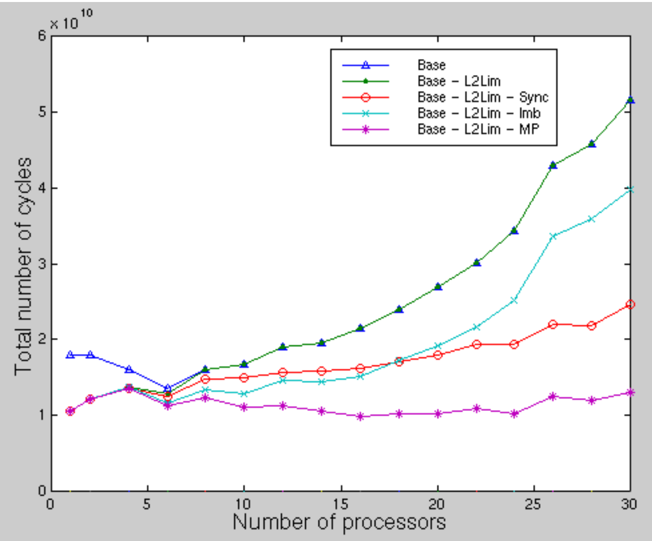

Figure 6: Estimation of the scalability bottlenecks in T3dheat.

are balanced and disjoint, there will be enough caching space with 10 processors (40 Mbytes / 4 Mbytes). This is consistent with *Scal-Tool*'s prediction that the *L2Lim* effect is negligible past 8 processors.

It is hard to use use SGI's tools to validate the curves in Figure 6. *Speedshop* cannot measure *L2Lim*. Furthermore, it cannot separate *Sync* from *Imb* because the time spent in the synchronization routines also includes load-imbalance spinning. We can, however, use *speedshop* PC sampling to validate the total *MP=Sync+Imb* effect. *Speedshop* measures the cycles spent in barrier-related functions (*mp_barrier()*, *_nthreads()*, and *mp_lock_try()*) and load imbalance functions (*mp_slave_wait_for_work()* and *mp_master_wait_for_slaves()*). The measurement is compared to the *MP* value estimated by *Scal-Tool*. The two curves are shown in Figure 7. The figure is organized like Figure 6. It shows that the *MP* cost estimated by *Scal-Tool* is remarkably similar to the one measured by *speedshop*.

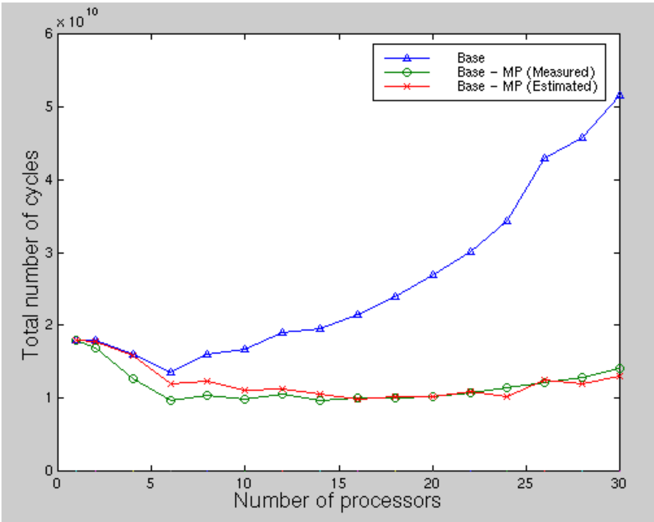

Figure 7: Validation of the model for T3dheat.

4.2 Hydro2d

The speedup curve for Hydro2d is shown in Figure 8. It shows that the Origin 2000 delivers only modest speedups. To understand the speedups, we use *Scal-Tool* to break down the total measured cycles into the estimated components (Figure 9). The figure is organized like Figure 6.

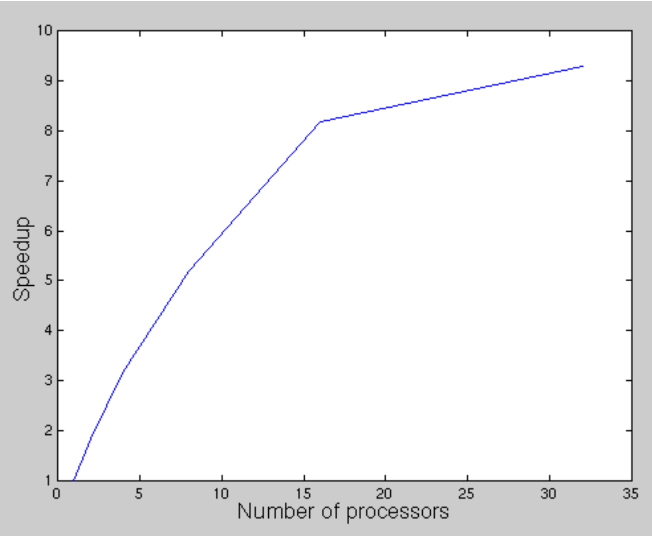

Figure 8: Speedups for Hydro2d.

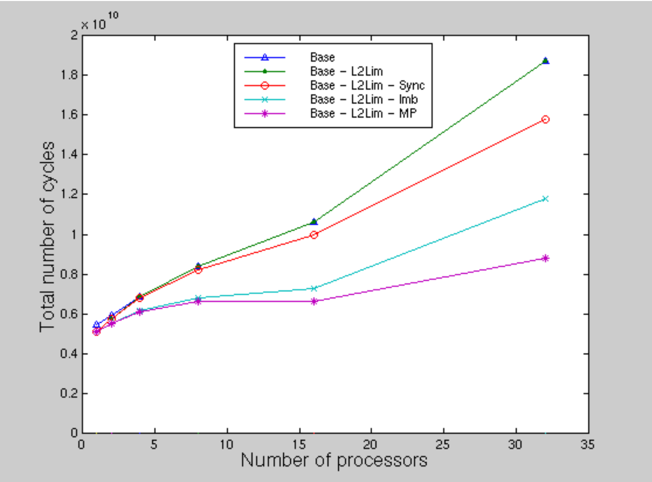

Figure 9: Estimation of the scalability bottlenecks in Hydro2d.

In the figure, the *Base-L2Lim* curve overlaps completely with the *Base* curve after 2 processors. Consequently, the effects of limited caching space are negligible. However, if we examine the *Base-L2Lim-Sync* and *Base-L2Lim-Imb* curves, we see that this application suffers from significant load imbalance. A more balanced load would increase the speedup significantly. Synchronization is not as costly, although it also induces some overhead. The figure shows that, without load imbalance or synchronization overhead, the application would about double its speed for 32 processors.

To validate the estimated *L2Lim* effect, we use *ssusage* to measure the data set size of Hydro2d. The value measured is 10.3 Mbytes. Consequently, assuming balanced and disjoint per-processor working sets, the effect of limited caching space vanishes at 2-3 processors, which agrees with *Scal-Tool*'s prediction.

The comparison of MP cost measured by *speedshop* and predicted by *Scal-Tool* is shown in Figure 10. Overall, the figure shows that the estimated and the measured *MP* overhead are again very similar. For 32 processors, the predicted and the measured *Base-MP* curves differ by only 9% of the accumulated cycles of all processors.

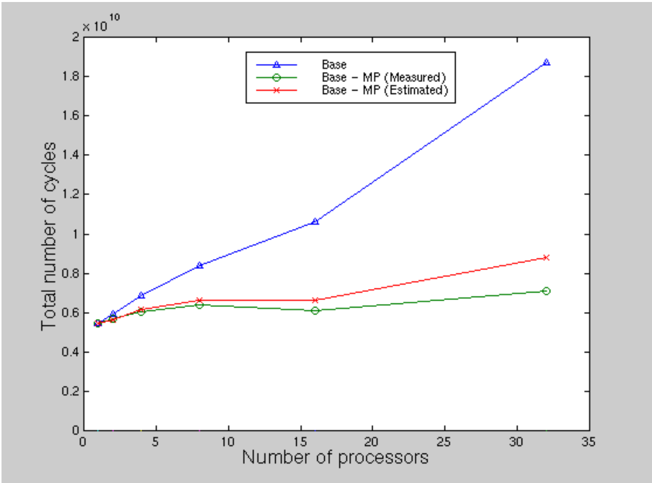

Figure 10: Validation of the model for Hydro2d.

4.3 Swim

The speedup curve for Swim is shown in Figure 11. It shows that the Origin 2000 delivers very good speedups. To understand the speedups, we use *Scal-Tool* to break down the total measured cycles into the estimated components (Figure 12). The figure is organized like Figure 9.

In the figure, the *Base-L2Lim* curve overlaps completely on top of the *Base* curve. Once more, the effects of limited caching space are negligible. The figure also shows that, of the multiprocessor effects, load imbalance dominates by far over synchronization. Therefore, it is largely load imbalance the effect that prevents Swim from reaching near linear speedup. The fact that the speedup for 32 processors is nearly 25 implies that this load imbalance is of modest magnitude. As a result, we will see next that it is hard to quantify exactly.

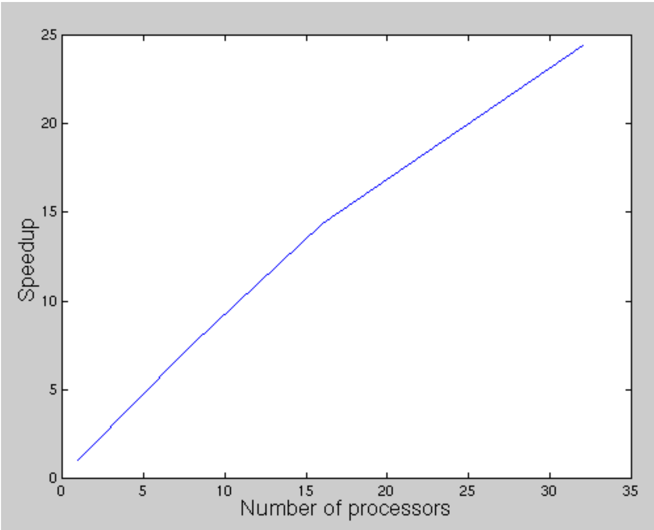

Figure 11: Speedups for Swim.

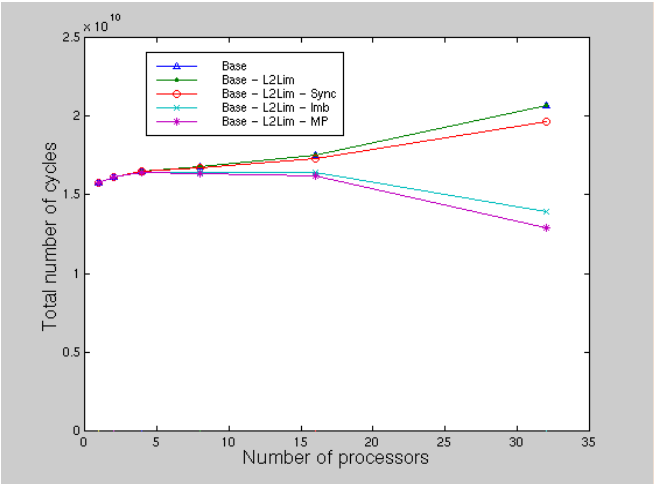

Figure 12: Estimation of the scalability bottlenecks in Swim.

To validate that the *L2Lim* effect is negligible, we use *ssusage* to measure the data set size of Swim. The value measured is 16.2 Mbytes. Consequently, assuming balanced and disjoint per-processor working sets, few processors are required to provide enough caching space.

The comparison of MP cost measured by *speedshop* and predicted by *Scal-Tool* is shown in Figure 13. The figure shows that, while until 16 processors, estimated and measured curves agree, they diverge for 32 processors. For 32 processors, the predicted and the measured *Base-MP* curves differ by 14% of the accumulated cycles of all processors. This difference is due to presence of non-synchronization data sharing in the program and the higher difficulty of quantifying load imbalance in applications with speedups close to linear. Hopefully, with an extension to *Scal-Tool* to estimate the effect of data sharing, the differences between the curves could be reduced.

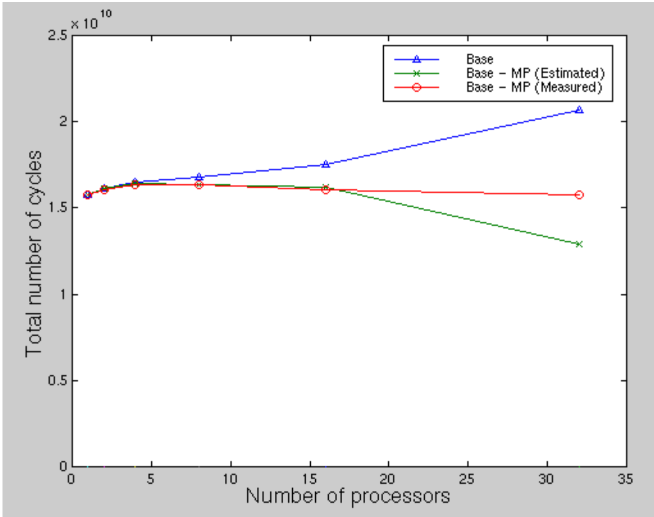

Figure 13: Validation of the model for Swim.

5 Related Work

Mathematical models, software-based simulators, performance tools, and empirical performance models, all provide ways to analyze the scalability of applications on DSM machines.

Mathematical models have been widely used. For example, they have been used to model the effect of load imbalance [5], the tradeoff between speedup and efficiency [4], and the contention in shared-memory machines [6]. While they are fast, they use simplified models, often with assumptions that restrict their accuracy and their applicability to real machines.

Software-based simulators simulate simplified machine models in detail. They tend to be more accurate than mathematical models. The applicability of the simulation results to real machines depends on how detailed the models being simulated are. Often, if significant levels of detail are required, software-based simulation approaches tend to be quite slow [2, 9, 13, 15, 23].

Performance tools that come with the machine tend to be useful. However, they may also have problems. First, they are often little integrated and are time-consuming to run. Another problem of some tools is that their output gives information that is too low level. For example, *perfex* outputs the number of data and instruction misses in the caches and the number of TLB misses. Programmers rarely know how to relate the numbers to their code's scalability bottlenecks. Another problem is that the tools often output numbers that are not directly relatable to the total execution time of the code. For example, they may generate the number of remote invalidations, but do not tell how many cycles they actually cost. Finally, it is typically impossible to evaluate a different architecture. For example, it is impossible to measure the misses if the L2 cache doubled in size.

Empirical performance models use performance tools to generate higher-abstraction information. These models can be very useful for programmers because they can often isolate the performance problems of their code. In addition, they often provide some ways to evaluate a different architecture. One such tool is the one proposed by Lubeck *et al* [12, 24], also based on CPIs, which predicts the performance improvement factors of the Origin2000 over the PowerChallenge architecture. Our tool is different from theirs, among other things, in that they model a uniprocessor, while we model a multiprocessor machine.

6 Conclusions and Future Work

In this paper, we presented *Scal-Tool*, a tool that isolates and quantifies scalability bottlenecks in parallel applications running on DSM multiprocessors. The inputs to *Scal-Tool* are measurements from hardware event counters in the processor. The scalability bottlenecks currently quantified include insufficient caching space, synchronization, and load imbalance. Major advantages of the tool are that it is relatively inexpensive to run, it is integrated, and provides ways to experiment with changes in different machine parameters.

We hope that *Scal-Tool* is useful to programmers early in the game in the development of an application. It is possibly unrealistic to expect the tool to quantify with high accuracy the cost of each bottleneck. However, we feel that pinpointing and roughly quantifying the bottlenecks is already a nice service to the programmer.

Work in progress includes extending *Scal-Tool* to incorporate the effect of true and false sharing. This extension should make the tool more accurate for some applications. Other work in progress includes testing the tool for large numbers of processors and experimenting with more applications.

Acknowledgments

We thank Yong Luo, Harvey Wasserman, and their colleagues for their feedback. We also thank the referees and the graduate students from the I-ACOMA group. Finally, we thank Dave Nicol for his help with the paper.

References

[1] T. Brewer and G. Astfalk. The evolution of the hp/convex exemplar. In *Proceedings of the COMPCON Spring 97: Forty-second IEEE Computer Society International Conference*, February 1997.

[2] D. Burger and T. Austin. The SimpleScalar Tool Set Version 2.0. Technical Report TR-1342, Computer Sciences Department, University of Wisconsin-Madison, May 1997.

[3] Data General. *The NUMALiiNE System. http://www.dg.com.*

[4] D. Eager et al. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(1), 1989.

[5] A. Eichenberger and S. Abraham. Modeling load imbalance and fuzzy barriers for scalable shared-memory multiprocessors. In *Proceedings of the 28th Hawaii International Conference of System Sciences*, January 1995.

[6] M. Frank et al. Lopc: Modeling contention in parallel algorithms. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[7] E. Hagersten and M. Koster. WildFire – A Scalable Path for SMPs. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann Publishers, San Francisco, CA, 1996.

[9] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

[10] J. Laudon and D. Lenoski. The sgi origin: a cc-numa highly scalable server. In *International Symposium on Computer Architecture*, June 1997.

[11] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[12] O. Lubeck, Y.Luo, H. Wasserman, and F. Bassetti. An empirical hierarchical memory model based on hardware performance counters. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, July 1998.

[13] V. Pai, P. Ranganathan, and S. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 72–83, February 1997.

[14] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[15] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Fast and Accurate Multiprocessor Simulation: The SimOS Approach. In *IEEE Parallel and Distributed Technology*, volume 3, Fall 1995.

[16] A. Sen and M. Srivastava. *Regression Analysis*. Springer-Verlag, New York, 1990.

[17] Silicon Graphics Inc. *fetchop man pages*.

[18] Silicon Graphics Inc. *perfex man pages*.

[19] Silicon Graphics Inc. *speedshop man pages*.

[20] Silicon Graphics Inc. *ssusage man pages*.

[21] The Standard Performance Evaluation Corporation. The SPEC95FP Suite. URL: http://www.specbench.org.

[22] J. Torrellas, M. Lam, and J. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. In *IEEE Trans. on Computers*, pages 651–663, June 1994.

[23] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

[24] H. Wasserman, O. Lubeck, Y. Luo, and F. Bassetti. Performance evaluation of the sgi origin2000: A memory-centric characterization of lanl asci applications. In *Proceedings of Supercomputing 1997*, November 1997.

[25] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Proceedings Supercomputing '96*, November 1996.