A Limit Study of Local Memory Requirements Using Value Reuse Profiles

Andrew S. Huang and John P. Shen
{ahuang,shen}@ece.cmu.edu

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA, 15213

(412) 268-3601

Abstract

Modern high-performance microprocessors are devoting more and more resources to the problem of the von Neuman bottleneck. In this limit study, we measure the bare minimum amount of local memories that programs require to run without delay. Our measurements are made by using the Value Reuse Profile, which contains the dynamic value reuse information of a program's execution, and by assuming the existence of efficient *memory systems. The results show that the group of 16 benchmarks we use require considerably less memory than what a typical superscalar microprocessor has. We also measure the amount of performance improvement that is possible in the presence of an* autonomous *memory system. For the DEC Alpha 21064, this figure ranges from 15% to 102%. The results provide motivation for the development of more effective memory management policies.*

1.0 .Introduction

Memory has always been a bottleneck in computer systems. This is because the memory in a typical computer system is unable to meet the latency and bandwidth requirements of its processor. The problem is made worse by two current trends. Advances in semiconductor technology and instruction pipelining make it possible for processors to run at higher clock rates. This increases the relative latency of memory access. At the same time, improvements in microarchitecture designs are making it possible for processors to issue multiple instructions per cycle. More instructions executed per cycle translates into a greater memory bandwidth requirement. The most popular response to these trends is to increase the number of levels of caches and to increase the size of the caches. For example, the recently introduced Alpha AXP 21164 [16] runs at 300 MHz and can issue four instructions per cycle. It has two levels of on-chip caches, which together account for 80% of the chip's transistor count. It is not clear how much of that cache is really needed to hold programs' working sets and how much is needed to make up for the inefficiencies of the traditional memory hierarchy.

This paper makes four major contributions. First, the fundamental issues that affect memory system performance are identified and discussed. Second, an autonomous memory system is proposed which provides the basis for evaluating the performance of existing memory systems. It represents a theoretical limit that may or may not be attainable. Programs running on a sufficiently large autonomous memory system are said to achieve an effective memory latency of zero cycles. Third, the Value Reuse Profile (VRP) is introduced. The VRP captures the dynamic value reuse characteristics of a program and can be used to compute the minimum amount of memory needed to achieve a specified level of performance. Finally, experimental results are presented which show that, on a modern RISC processor such as the DEC Alpha AXP 21064, some programs can run up to twice as fast in the presence of an autonomous memory system.

The rest of the paper is structured as follows. Section 2.0 provides the context for the paper and discusses the issues that affect memory system performance. Section 3.0 describes the construction of the Value Reuse Profile. Section 4.0 describes the use of the VRP in computing the lower bound on physical dimensions of the memory system. In Section 5.0, we describe our implementation of the VRP software. Our initial experimental results are shown in Section 6.0. Finally, we conclude in Section 7.0 and outline directions for future research in Section 8.0.

2.0 Background

In this section, we introduce some of the concepts and terminology used in the rest of the paper. We assume a hierarchical memory system in which each level of the hierarchy that is farther away from the CPU is also slower in speed [17]. The **local memory** consists of the levels in the memory hierarchy between the CPU and the main memory, exclusive. Typically, it includes high speed memories such as register files and caches.

A program's execution can be viewed as a dataflow graph in which each node represents a computation (performed, for example, by an arithmetic instruction of a CPU) and each arc is a value that is produced by one computation and consumed by another computation. We refer to these nodes as the **useful instructions**. When the program is executed on a von Neuman machine, additional instructions will be executed. These include instructions that manage the movement of values in the memory system. For example, loads and stores move values between the register file and the main memory. The move instructions move values within the register file. We call these **memory instructions**.

We define the **memory overhead** for a program running on a particular processor-memory system as the additional machine cycles incurred due to the execution of memory instructions. The **memory penalty** consists of additional machine cycles due to any additional delays, such as cache miss penalty, that are caused by executing memory instructions. The **effective memory latency** (EML) for a program on a processor-memory system is the sum of the memory overhead and the memory penalty, divided by the total number of value consumptions. Thus the EML, measured in number of cycles per value, measures the cost per value referenced for a particular combination of program and memory system. This will be our metric for measuring memory system performance.

The performance of a program on a processor-memory system and its EML are strongly influenced by the following four fundamental factors:

- **Predictability**. This refers to the ease with which references to values can be predicted. In general, numerical programs have higher levels of predictability than integer programs. This is because numerical programs generally have more predictable control flow.
- **Value Reuse**. This refers to the reuse characteristics of the data in a program. Some values are referenced frequently in a short length of time, while others are referenced occasionally over long periods of time. The reuse characteristic determines the amount of local memory that is needed to store live values.
- **Management Policy**. This refers to the way in which storage allocation is performed by the compiler and the policy based on which data is moved up and down the memory hierarchy. It includes the statically scheduled movement of data using memory instructions. It also includes the dynamic movement of data into the cache and the replacement of a cache line on a cache miss.
- **Physical Dimensions**. This refers to the physical size of each level of the memory hierarchy and the bandwidths of the inter-level connections. A local memory that is too small to hold the frequently referenced data will stall the processor and increase the EML. A local memory that is too big is a wasted resource that could be better spent elsewhere.

To a great extent, the predictability of a program determines the most suitable memory management policy. Programs that have a high level of predictability can benefit considerably from better management policies such as compiler control of data movement through the cache [3][5][7][8][13][19][21] and better storage allocation [2][4]. They can also benefit from locality enhancing transformations that result in more efficient usage of the available local memories [11][18][24].

In this study, we factor out the issues of predictability and management policy to focus on the relationship between the value reuse characteristics of programs and the required physical dimensions of the local memory. In particular, we are interested in using the VRP to find the lower bound on the physical dimensions of the memory hierarchy that is necessary in order for a program to achieve a certain level of performance. The best possible performance is achieving EML of zero. Predictability is factored out by using program traces to provide perfect knowledge of memory references. Management policy is factored out by looking beyond **variables** (which are the artifacts of the compiler's storage allocation) at **values** and by assuming the existence of management policies that can guarantee zero memory penalty in the presence of sufficient memory.

We propose two types of memory systems. In the **efficient memory system**, the memory penalty is guaranteed to be zero when a program is run on a system with sufficiently large physical dimensions. The bounds computed using this memory system tells us how little local memory is required if current memory systems could be made more efficient. By efficient, we mean that caches are not used to hold dead values or values that will be overwritten before they are used, etc. In the **autonomous memory system**, both the memory penalty and the memory overhead are guaranteed to be zero when a program is run on a system with sufficient memory. This memory system represents the best possible memory system that can ever be built. Programs running on a sufficiently large autonomous memory system will have EML of zero.

Previously, Grimsrud et al. [14] proposed using the **locality surface** (somewhat like the VRP plot discussed in Section 3.3) to evaluate the reference streams generated by synthetic models. The locality surface is a plot of probability versus stride and distance (time between references). Thus it represents a summary of the spatial and temporal locality of a reference stream. Our work is different because we factor out the compiler's storage allocation scheme to look at the temporal locality of values. McNiven and Davidson [20] studied memory referencing behavior for the purpose of developing new memory management policies and memory structures. They used a trace processing technique called **flattening** to remove the effects of compiler storage allocation. By doing so, they were able to obtain summary statistics on values. Our work use a technique similar to flattening. However, we collect detailed information on values over the course of the program's execution time. This gives us a profile that shows variations in referencing behavior over time.

3.0 Value Reuse Profile

The VRP is a complete specification of the cycle times at which values are defined and used. As soon as a value is defined, it needs a storage location. Prior to its next use, the value must be delivered from its storage location to the CPU. By specifying the times and durations during which values must be stored, the VRP specifies the memory requirements of the program that generated it. Before describing the construction of the VRP, we briefly review the basic concepts regarding values.

3.1 Values

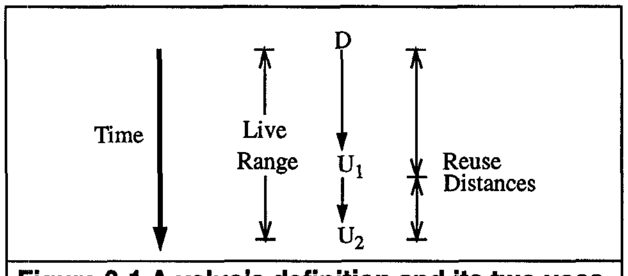

Figure 3-1 A value's definition and its two uses.

A value is defined once and used one or more times, after which, it is **dead**. The distance between the definition and the last use is known as the **live range** of the value. The distances between consecutive references to the value are the **reuse distances**. The **next use distance** of a live value at a point in time is the distance (in number of cycles) to its next reference. Figure 3-1 shows a value with two uses.

3.2 Construction

Information about value reuse can be gleaned from a trace. However, the trace by itself contains only the relative order in which values are defined and used. To obtain the VRP, we feed the trace into a timing simulator, which timestamps each value definition and use (in number of clock cycles since the program started executing.) Figure 3-2 shows the setup.

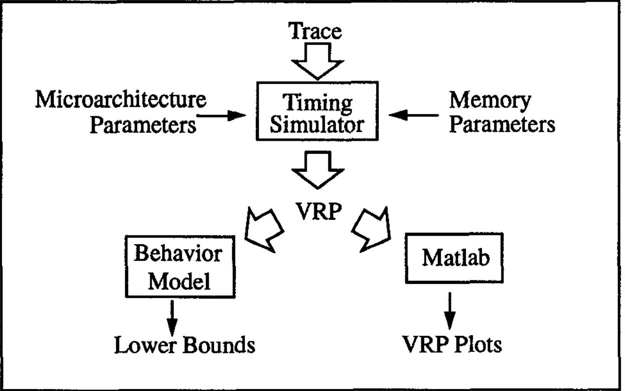

Figure 3-2 Construction and uses of VRP.

The timing simulator takes two major groups of parameters. Microarchitecture parameters specify the machine characteristics that should be used. This can include instruction latencies, issue policies (such as in-order or out-of-order issue), and number of functional units. Memory parameters specifies memory latencies, cache sizes, and memory models. The two supported memory models that are relevant to this paper are the efficient memory system and the autonomous memory system.

Each combination of parameters can result in a different VRP even when the same trace is used. Together, the parameters allow us to generate VRPs for a universe of processor-memory systems that span from the conventional to the very idealistic.

3.3 Visualization

We can plot the Value Reuse Profile (VRP) as a three-dimensional surface of $n = f(t,x)$ where $f(t,x)$ is the number of values live at time t with a next use distance of x cycles (See Figure 3-4.) Such a plot not only shows the number of values live at each point in time, it also reveals how many of the live values will soon be referenced.

Figure 3-3 illustrates the construction of a VRP plot by showing how one value contributes to the plot. The value is defined at $t = 1$ and is first used at $t = 6$. Therefore at $t = 1$, it is live and has a next use distance of 5 cycles ($t = 1, x = 5$). At $t = 2$, it is still live, and has a next use distance of 4 cycles. At $t = 6$, the value is used in a computation ($t = 6, x = 0$). Its next use occurs at $t = 9$, which is 3 cycles away ($t = 6, x = 3$.)

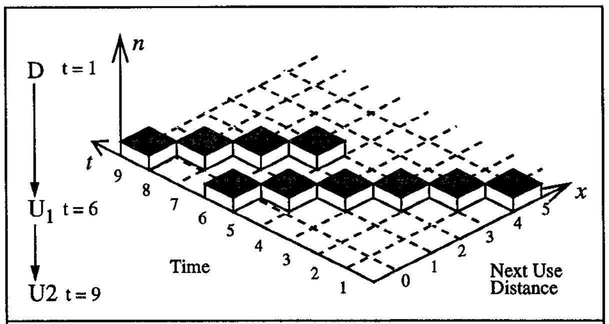

Figure 3-3 The contribution of a value to the Value Reuse Profile.

When the contributions from all values in the VRP are added up, the result is a VRP surface that resembles Figure 3-4. This is a VRP plot for a version of *tomcatv* that has been modified to execute only four iterations. Note that macro characteristics such as the number of iterations of the outer loop are clearly visible. Many of these VRPs exhibit interesting and insightful terrains.

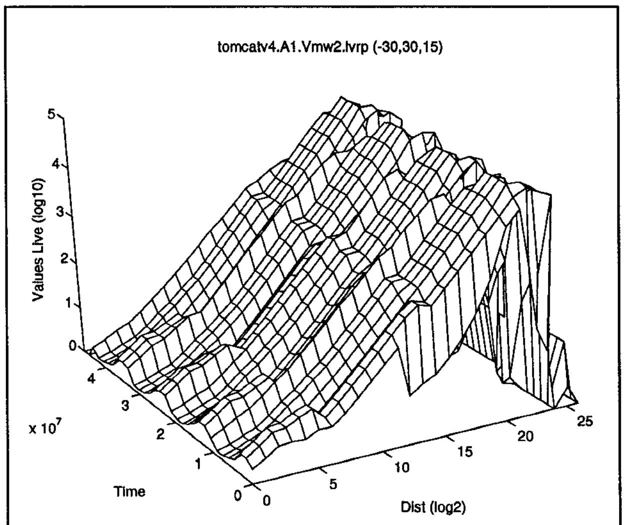

Figure 3-4 Sample VRP, from *tomcatv*.

4.0 Lower Bounds on Physical Dimensions

Given a VRP, it is possible to compute the minimum amount of local memory that is necessary in order to achieve the performance level specified by the VRP. Since one can arbitrarily decrease the size of the memory at level i at the expense of increasing the bandwidth requirement between the memory levels i and i+1, it is insufficient to talk about size alone. We use the term physical dimension to mean both the size and bandwidth for each level in the memory hierarchy. In this section, we present two management policies to help us compute this lower bound on physical dimensions. Both policies guarantee zero memory penalty when the memory system is at least as large as the computed lower bound.

Both policies assume a memory hierarchy with small, high-speed memory close to the CPU and large, slow-speed memory far from the CPU. The highest level in the memory hierarchy is the one next to the CPU. The lowest level is the one farthest away from the CPU.

In our study, values are stored in the various levels of the memory hierarchy and can be moved between levels. Unlike in conventional cache memories, the principle of inclusion, i.e. that lower levels must contain all of the data held in higher levels, is not imposed.

4.1 Simple Policy

The Simple policy is based on the VRP plot. A value is kept in a particular level of the hierarchy based on how far in the future the value will be used. At any given time, all values with next use distance x such that $x_{i-1} < x \leq x_i$ are stored in M_i, level i of the memory hierarchy. Values that will be needed soon are stored closer to the CPU than those that will be needed later on. As time passes on, and the next use distance of a value decreases to a critical point, the value is automatically moved to the next higher level in the hierarchy. By the time that a value is needed for a computation, it has been moved to the level in the memory hierarchy that is closest to the CPU. At any given time, each value can be in exactly one level. In other words, values are not allowed to be in the register file and the cache at the same time.

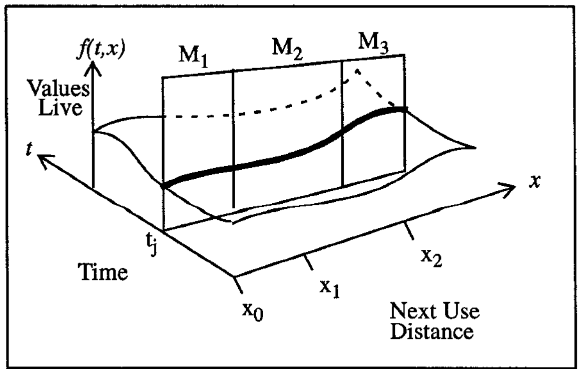

Figure 4-1 Obtaining memory size requirements from the VRP.

Given such a management policy, it is possible to find the lower bound on the size and bandwidth requirement of each level of the memory hierarchy by examining a program's VRP plot. At any given time t_j, the number of live values can be obtained as follows. The intersection of the VRP plot with the plane $t = t_j$ produces a curve (the bold line in Figure 4-1).

The area under this curve is the total number of values live at t_j. Thus the size requirement of M_i can be found by integrating over a portion of the curve (from x_{i-1} to x_i) and finding the maximum of such integrals along the time axis.

The bandwidth requirements can be found in a similar way. The intersection of the VRP surface with the plane $x = x_{i-1}$ produces a curve (bold line in Figure 4-2). The height of this curve at a point in time is the number of values that are transferred from M_i to M_{i-1} at that time. The area under the curve is the total number of values transferred over the course of the program's execution. To find the maximum bandwidth requirement, it is only necessary to find the highest point on this curve.

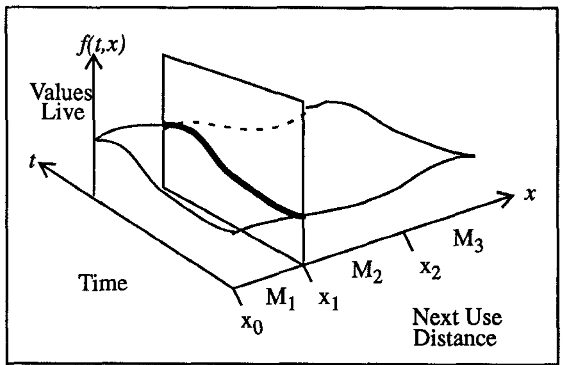

Figure 4-2 Obtaining bandwidth requirements from the VRP.

The weakness of the Simple policy is its potential poor utilization of memory. Values are assigned to levels in the memory hierarchy based only on their next use distances. This can result in many unoccupied spaces in the highest levels of the memory hierarchy while values with large next use distances are written to lower levels.

4.2 Pack Policy

In the Pack policy, values are stored in the highest level of the memory hierarchy first. Only when the highest level fills up are values stored in the next lower level.

When inter-level transfer of values are needed, the model always attempts to space the transfers apart as much as possible to minimize the instantaneous maximum bandwidth requirement of the program.

Using this policy, we can compute the minimum amount of inter-level bandwidth required given the capacity (or size) of each level.

We have found that the bounds computed using the Pack policy are always lower than those computed using the Simple policy. This paper will only present data obtained using the Pack policy.

5.0 Implementation

The software used in our data collection and visualization consists of the three components shown in Figure 3-2. Using the ATOM Tools [23], we instrument the benchmarks to produce instruction and data traces, which are fed into the timing simulator for the DEC Alpha 21064. The simulator takes into consideration data dependence, structural dependence, issue policy, and cache misses to produce timing information for all instructions executed. The output of the simulation is a VRP, which can be used to compute lower bounds on physical dimensions or to produce a VRP plot. In the rest of this section, we discuss some issues involved in our software implementation.

5.1 Tracking Values

A value *v* is created when an instruction performs a computation and writes the result *v* to a destination register, say R3. The only way to know that *v* dies is when a later instruction overwrites R3, because there is no longer a way to reference *v*.

However, it is possible for an intervening *move* instruction to copy *v* into some register R4. In this case, *v* is dead only when both R3 and R4 are overwritten by later instructions. A related issue is the computation of reuse distances. After *v* is referenced through R3, the next reference to *v* may be made via R3 or R4. If the trace processing software only looks for references to R3, the result can be incorrect. In more complicated cases, a value can simultaneously reside in several registers as well as several memory locations. Our software tool solves these problems by mapping registers and memory locations to a value space, where information about values is kept.

5.2 Other Tools

In addition to the VRP module, another software tool called VMW (Visualization-based Microarchitecture Workbench) [9] has been developed at Carnegie Mellon University. Using VMW, a timing simulator for an arbitrary machine can be automatically "compiled" from a set of machine description files. This is how the Alpha 21064 simulator is built. A module to construct the VRP is then added to the VMW generated simulator. We use the MAT-

LAB software to produce VRP plots. In this paper, we use the Alpha AXP 21064 as the experimentation vehicle, due to the availability of the ATOM profiling tools from DEC and actual 21064 based systems for physical validation.

6.0 Experimental Results

This section presents two major groups of results for a set of 16 benchmarks. Only read traffic is considered, since it has the bigger impact on performance.

The first group contains the lower bounds on the bandwidth of the memory hierarchy that is necessary to achieve zero memory penalty given the capacities of the Alpha 21064's local memory. The lower bound on the required bandwidth of the memory system is found by measuring the peak bandwidth requirements of the programs. We also do sensitivity analysis on the effects of varying the capacities. Our experiments consider reuse only. They do not account for the traffic generated the first time that an initialized static variable is read from memory, nor do they account for program input.

The second group contains memory system statistics, which includes the efficiency of bandwidth usage on the Alpha 21064 and the speedup that can be expected if it were possible to achieve zero-cycle EML. These data are for an Alpha 21064 system, a dual issue microprocessor with 8K each of onchip instruction and data cache. These Level 1 caches are direct mapped and have 32-byte lines. For our experiments we assume that there is a 256K Level 2 data cache with 2-way set associativity and 32-byte lines. The L2 data cache has a write allocate policy. In the remainder of this paper, the term cache is used to mean data cache.

The simulations are run on DEC 3000 Models 400 and 500. These are Alpha 21064 systems with 512K Level 2 combined cache and 64M RAM. They run at 133MHz and 150MHz respectively. On the Model 400, the simulator (in which VRP module is embedded) run at the rate of approximately 7,000 simulated cycles per second.

Table 6-1 lists the 16 benchmarks used in the experiments. They have been divided into three groups to simplify data presentation. The first group is the **Integer** benchmarks, which consists of two benchmarks from SPECINT 92, two graphics programs and a quicksort. The next group is **Nasa7**, which consists of six benchmarks from SPECFP 92's NASA7 benchmark, which has been broken up to reduce simulation time and to expose characteristics of the individual programs. The last group is **SpecFP**, which contains other floating-point benchmarks from SPECFP 92. The last column in the table is the number of cycles the programs take to execute on a real memory system. These numbers are obtained from the timing simulator, and they match quite closely with actual run times on the DEC 3000 systems.

Table 6-1 Benchmark descriptions

	Benchmark	Data/Iterations	Run time (cyc)
Integer	cjpeg	128x128 BW image.	9,024,934
	compress	1 iteration. (SPEC92 uses 20.)	119,446,305
	mpeg	4 frames w/ fast dithering.	17,557,875
	quick	5,000 elements.	1,090,390
	xlisp	5 queens (SPEC92 uses 9.)	20,733,986
Nasa7	btrix	1 iteration. (NASA7 uses 20.)	119,837,396
	cholsky	5 iterations. (NASA7 uses 200.)	70,803,957
	emit	1 iteration. (NASA7 uses 10.)	97,206,623
	fft	2 iterations. (NASA7 uses 100.)	46,143,645
	mxm	2 iterations (NASA7 uses 100.)	25,273,771
	vpenta	5 iterations (NASA7 uses 400.)	43,899,386
SpecFP	doduc	Tiny input from SPEC92.	98,732,815
	ear	Modified short input from SPEC92	260,503,257
	hydro2d	Short input from SPEC92.	10,566,371
	swm256	5 iterations (SPEC92 uses 1,200).	81,158,983
	tomcatv	4 iterations (SPEC92 uses 100.)	61,636,976

6.1 Computation of Lower Bounds

In this subsection, we measure the amount of bandwidth that is required to run the benchmarks with no memory penalty on a memory hierarchy with the same sizes as that of the DEC Alpha 21064. Table 6-2 lists the capacities of the Alpha 21064's memory system.

Table 6-2 21064 memory system parameters.

	M_1	M_2	M_3
Description	Register File	L1 Cache	L2 Cache
Size (bytes)	512	8K	256K
Size (# values)	64	1,024	32,768

The 21064 has a total of 64 registers, 32 in the floating point register file and 32 in the integer register file. The sizes of the L1 and L2 caches (M_2 and M_3) are computed by assuming that values are 64 bits wide. If values are 32 bit wide instead (which is true for many of the benchmarks), the sizes of the 21064's local memory would be twice the numbers stated in the table.

Table 6-3 lists the peak bandwidths required by the programs. They are computed from the VRPs using the above parameters. For comparison the last row of the table lists the actual bandwidths for the 21064's memory hierarchy. The 21064 can issue two instructions every cycle, each of which can read two values from the register file, for a total of four value reads per cycle. It can execute one load per cycle, hence the transfer rate of one between the register file and the L1 cache. The bandwidths into the L1 and the L2 cache are computed with the assumption of 64 bit wide values. The L1 can read in 128 bits of data in four cycle, or two values every four cycles. The L2 can read 128 bits of data in seven cycles, or two values every seven cycles.

Table 6-3 Peak bandwidth required by programs (efficient memory).

Program		Bandwidth from lower level(val/cyc)			
		CPU	Reg File	L1 cache	L2 cache
Integer	cjpeg	4.000	0.104	0.024	0.000
	compress	3.000	0.105	0.060	0.024
	mpeg	5.000	0.289	0.005	0.000
	quick	3.000	0.110	0.089	0.000
	xlisp	3.000	0.084	0.024	0.000
Nasa7	btrix	4.000	0.272	0.132	0.029
	cholsky	4.000	0.244	0.139	0.046
	emit	4.000	0.177	0.068	0.057
	fft	4.000	0.340	0.332	0.170
	mxm	4.000	0.316	0.316	0.004
	vpenta	4.000	0.247	0.245	0.069
SpecFP	doduc	4.000	0.246	0.014	0.000
	ear	6.000	0.083	0.000	0.000
	hydro2d	5.000	0.352	0.119	0.000
	swm256	4.000	0.293	0.289	0.253
	tomcatv	4.000	0.449	0.446	**0.392**
21064		**4**	**1**	**0.5**	**0.286**

The entries in this table specify the maximum bandwidth requirement of each program. For example, *cjpeg* (the first program in the first group) has a bandwidth requirement of 0.104 values per cycle between L1 cache and the register file. This means that during the time of greatest traffic intensity between L1 cache and the register file, *cjpeg* needs to be able to transfer one value from L1 cache to the register file every 9.6 cycles (9.6 = 1 / 0.104) in order to meet the value usage deadlines specified in the VRP

One value per 9.6 cycles is the *peak* bandwidth requirement. The bandwidth requirement averaged over the course of the program's execution is just one value per 40 cycles (see Table 6-4), about one fourth of the peak.

When we first examined this table, we were baffled by the fact that *mpeg*, *ear*, and *hydro2d* can require more than four values per cycle when only two instruction can be issued per cycle, and each can only consume up to two values. It turns out that this is due to a memory optimization performed by the DEC compiler, which can cause our software to register three value consumptions for each instruction executed under a rare situation. This special case arises infrequently and has no visible impact on other parts of our data.

Consider the last three columns, which lists the needed bandwidth into the register file, the L1 cache, and the L2 cache. For all benchmarks except *tomcatv*, the needed bandwidth is always quite a bit less than the actual bandwidth of the 21064. The entry in bold is the only one that is greater than the actual 21064 bandwidth. Approximately half of the benchmarks listed can store all of their intermediate values within the caches. These benchmarks are the ones with a zero L2 bandwidth requirement. One of them, *ear*, can fit all of its intermediate values within the register file and the 8K L1 cache.

For comparison, Table 6-4 shows the *average* bandwidth required by the programs. All average bandwidths are considerably less than the peak bandwidths.

Table 6-4 Average bandwidth required by programs (efficient memory).

Program		Bandwidth from lower level(val/cyc)			
		CPU	Reg File	L1 cache	L2 cache
Integer	cjpeg	1.192	0.025	0.007	0.000
	compress	1.076	0.048	0.030	0.008
	mpeg	1.162	0.019	0.002	0.000
	quick	1.234	0.045	0.026	0.000
	xlisp	0.785	0.039	0.009	0.000
Nasa7	btrix	0.720	0.123	0.061	0.019
	cholsky	0.735	0.117	0.081	0.025
	emit	0.957	0.055	0.035	0.002
	fft	0.972	0.102	0.100	0.091
	mxm	1.238	0.214	0.213	0.003
	vpenta	0.658	0.058	0.058	0.023
SpecFP	doduc	0.528	0.055	0.005	0.000
	ear	0.684	0.040	0.000	0.000
	hydro2d	0.681	0.068	0.014	0.000
	swm256	0.985	0.104	0.099	0.068
	tomcatv	0.923	0.117	0.116	0.068
21064		**4**	**1**	**0.5**	**0.286**

We now examine the effects of varying the capacities of each level of local memory on the maximum bandwidth requirements of the programs. For each level, the capacity is varied from one fourth to twice the capacity of the corresponding level in the 21064. Figure 6-1,Figure 6-2, and Figure 6-3 plots the required bandwidth versus capacity for the register file, L1 cache and L2 cache respectively. For the caches, the capacities are specified in both kilobytes and in number of values.

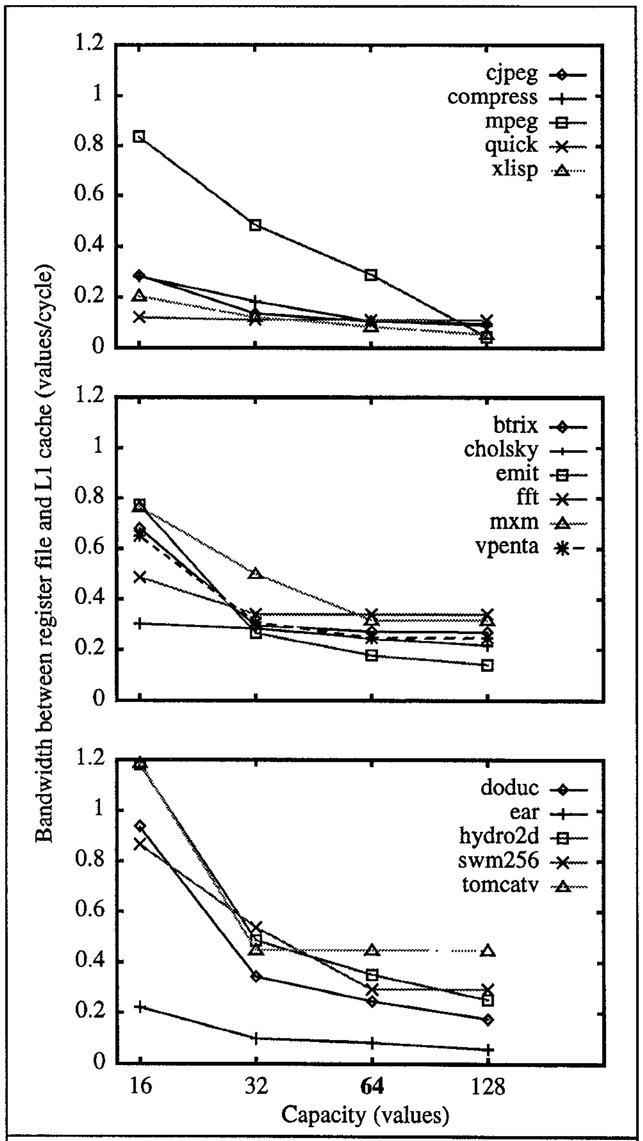

Figure 6-1 Peak bandwidth into register file.

The three plots of Figure 6-1 show the expected characteristics of decreasing bandwidth requirement when the capacity is increased. Most programs get the largest decrease in bandwidth requirement in going from a 16 to a 32 entry register file. Many of them stand to benefit from even larger register files.

For the L1 cache, the benefit of increasing capacity is not quite as dramatic. Most programs show only gradual decrease in bandwidth requirement with increasing capacity. The most notable exception is *mxm* (in the middle plot of Figure 6-2). The benchmark *mxm* is a blocked matrix multiply that operates on sections of two matrices. The two sections together contains 1,088 values, which will fit entirely in a 2,048 entry cache, but not in a 1,024 entry cache. This explains the abrupt drop in bandwidth requirement in going from a L1 cache that can hold 1,024 values to one that can hold 2,048 values.

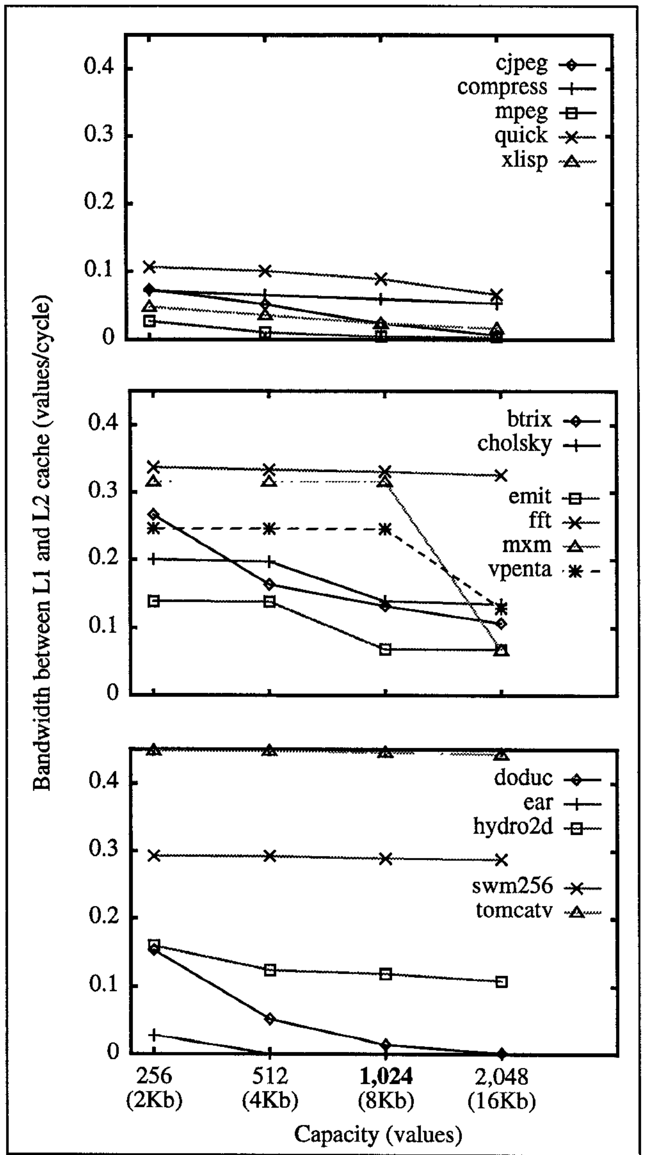

Figure 6-2 Peak bandwidth into L1 cache.

For the L2 cache (Figure 6-3), the first plot shows the **Nasa7** benchmarks. *compress* is the only **Integer** benchmark with nonzero bandwidth requirement. It is plotted along with two **SpecFP** benchmarks in the second plot. As this figure shows, most benchmarks have bandwidth requirements that drop off to zero as the size of the L2 cache is increased to 64K values. *swm256* and *tomcatv* are the only benchmarks with working sets larger than 64K.

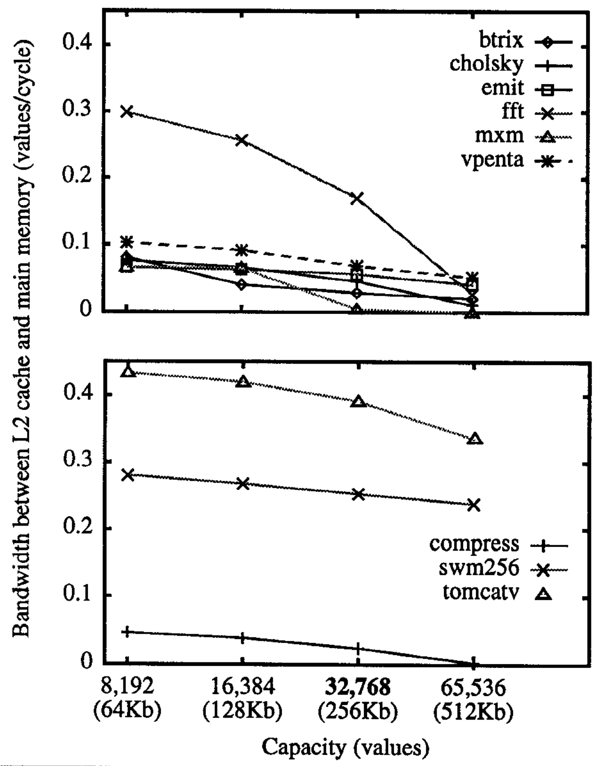

Figure 6-3 Peak bandwidth into L2 cache.

6.2 Memory System Statistics

What can we expect to gain from achieving zero-cycle EML? Table 6-5 lists some relevant statistics about the benchmarks' performance on the memory systems. The second column of the table is the speedup that can be obtained by executing on an autonomous memory system (with EML = 0) as compared to executing on the 21064's memory system. It is computed as $(t_r/t_a) - 1$, where t_r is the execution time on the real memory system (the 21064's memory system), and t_a is the execution time on the autonomous memory system. This represents the maximum amount of performance gain that is possible from improving the memory system and the compiler's storage allocation. Column three lists the actual EML, which measures the efficiency of the 21064's memory system in delivering needed values to the CPU. The fourth column is the average number of value references that occur for every memory instruction that is executed. Not surprisingly, programs that stand to benefit the most from the autonomous memory system are the ones with a small number of value references per memory instruction (*xlisp* and *compress.*)

Column five is the miss rate of the L1 cache. It shows that programs with high miss rates (*cholsky*, *fft*, *tomcatv*, and *vpenta*) generally have high EML as well. However, the opposite is not true. *xlisp* has an EML of about 0.6, but its miss rate is only 10%. Thus an insufficiently large (or poorly utilized) cache is a contributing factor to the poor performance of the memory system, but it is not the only factor.

The last column lists the bandwidth efficiency between the L1 cache and the L2 cache. We define the transfer of a value into L1 cache (from L2 cache) as useful if the value is then loaded into the register file before it is overwritten in the L1 cache. The bandwidth efficiency is the number of useful transfers divided by the total number of values transferred. As the column shows, half of the benchmarks listed have bandwidth efficiency of less than 50%. This means that over half of the values loaded into the L1 cache are either never used or are overwritten (due to cache line replacement policy or insufficient capacity) before they are used.

Table 6-5 Memory System Statistics

Program		Potential Speedup	EML cyc/val	#val/ #mem	L1 Misses	L1 BW Efficiency
Integer	cjpeg	32.5%	0.169	5.24	6.8%	39.3%
	compress	99.1%	0.510	3.48	20.0%	29.8%
	mpeg	27.3%	0.161	5.54	15.6%	25.9%
	quick	58.9%	0.285	4.09	1.6%	81.4%
	xlisp	101.6%	0.598	2.50	9.9%	44.1%
Nasa7	btrix	23.7%	0.251	4.02	31.8%	48.5%
	cholsky	64.7%	0.532	3.68	43.9%	41.9%
	emit	23.2%	0.169	5.39	7.1%	86.7%
	fft	67.2%	0.444	3.86	35.8%	56.8%
	mxm	46.8%	0.245	4.45	15.4%	70.7%
	vpenta	28.5%	0.300	4.63	42.8%	32.0%
SpecFP	doduc	14.7%	0.224	4.15	8.1%	56.5%
	ear	24.7%	0.237	4.25	2.2%	78.7%
	hydro2d	19.7%	0.218	4.99	16.0%	78.8%
	swm256	39.5%	0.266	4.59	20.3%	34.0%
	tomcatv	36.8%	0.286	4.42	35.3%	55.5%

We have placed the last two columns here to point out some of the characteristics of the traditional memory hierarchy, not to criticize the design of the 21064 in particular. Memory system design requires the consideration of many complex issues. In the case of the 21064, the designers have placed clock speed as a high priority. This explains the use of a small direct-mapped Level 1 cache, which can be expected to have higher miss rates and lower bandwidth efficiency than a cache of greater capacity and/or greater set associativity.

Table 6-5 shows that performance gain of up to 100% is possible with an autonomous memory system. In Table 6-6, we show the peak bandwidth required by the programs in order to achieve zero-cycle EML on an autonomous memory system. Once again, we have used the bold font at the places where a program need more bandwidth than is available from the 21064. We would expect the bandwidth requirement of the programs to increase because there are no longer any memory instructions to spread apart value references. Comparing this table against Table 6-3, we see that most programs have increased their bandwidth requirement by less than 100%. *tomcatv* has more than tripled its bandwidth requirements while *cholsky* and *fft* have also increased their bandwidth requirements significantly. However, most programs still require no more bandwidth than what the 21064 already has.

Table 6-6 Peak bandwidth required by programs (autonomous memory).

Program		Bandwidth from lower level (val/cyc)			
		CPU	Reg File	L1 cache	L2 cache
Integer	cjpeg	3.000	0.125	0.029	0.000
	compress	3.000	0.154	0.093	0.039
	mpeg	5.000	0.287	0.006	0.000
	quick	3.000	0.198	0.161	0.000
	xlisp	3.000	0.143	0.036	0.000
Nasa7	btrix	4.000	0.337	0.161	0.033
	cholsky	4.000	0.800	0.255	0.084
	emit	4.000	0.307	0.135	0.109
	fft	4.000	0.828	**0.807**	**0.418**
	mxm	4.000	0.510	0.500	0.006
	vpenta	4.000	0.361	0.359	0.100
SpecFP	doduc	4.000	0.366	0.016	0.000
	ear	5.000	0.107	0.000	0.000
	hydro2d	4.000	0.504	0.148	0.000
	swm256	4.000	0.472	0.436	**0.412**
	tomcatv	4.000	**1.404**	**1.391**	**1.226**
21064		**4**	**1**	**0.5**	**0.286**

7.0 Conclusion

Memory system performance is becoming an increasingly critical part of a program's performance. This has resulted in a significant amount of research into software and hardware techniques for improving memory systems performance. By introducing our concept of zero-cycle EML, we have provided a way to measure the maximum amount of performance gain that can ever be realized from such techniques. The potential can be considerable. According to Table 6-5 as much as 100% performance improvement is possible by improving the memory system alone.

We have presented a framework for a systematic approach to understanding the much talked about memory bottleneck problem. Four issues are involved in memory system performance: predictability, value reuse, management policy, and physical dimensions (size) of the memory system. By factoring out the issues of predictability and management policy, we are able to use value reuse information (provided by the VRP) to compute the minimum physical dimensions of the memory system that is required to achieve zero memory penalty. Our measurements show that the Alpha 21064 has more than the minimum memory required to achieve zero memory penalty for many programs, yet these programs have miss rates as high as 43%. This indicates that the problem isn't with the size of local memory, but with program predictability and/or memory management policy. Since many of these programs are floating-point applications with high degree of predictability, we can possibly draw the conclusion that memory management policy is one area to investigate in order to reduce the EML and achieve the potential speedup.

8.0 Future Directions

For expediency reasons, in this paper we have chosen the Alpha 21064 superscalar processor as the experimentation vehicle. The execution traces, used for generating all the VRPs and the experimental results on memory system performance, are all based on the Alpha 21064. Essentially the microarchitecture parameters are kept fixed in this study. The framework and associated software tools we have developed are general and can be applied to other architectures and microarchitectures. This is one possible direction of future research. The width of the machine and the aggressiveness of the microarchitecture can be varied to produce variations in the VRPs, which can in turn produce different results on memory system performance and requirements. In fact studies can be performed using an "infinite machine," in which the microarchitecture, or more specifically, machine resource constraints are not considered. The actual record of execution as well as the resultant VRP are strictly a function of the program characteristics and unaffected by any machine structural dependences. Such machine-independent results can characterize the asymptotic behavior of memory system performance.

While this paper primarily focuses on the analysis aspect, a great deal of future work can be done along the synthesis aspect. If the results of this paper can be generalized, they provide significant motivations for exploring better approaches to memory management policies. The results in Table 6-5 seem to indicate that current memory systems incur substantial EML overhead, and if EML=0 can be achieved, significant performance gains are possible. Furthermore, Table 6-3 implies that achieving EML=0 does not necessarily require unrealistic capacities and bandwidths for the physical dimensions of the memory hierarchy. Future work can reexamine current memory management policies to determine the reasons for their ineffectiveness and develop more effective policies to reduce the EML overhead. Recent techniques for data prefetching [6][7][10][21] can be viewed as specific attempts along this direction. Other more aggressive and perhaps more comprehensive approaches can potentially be developed.

9.0 Acknowledgment

The authors would like to thank the anonymous reviewers for the detailed reviews. We would also like to thank the Pittsburgh Supercomputing Center for the use of its DEC Alpha SuperCluster. This research is supported in part by an IBM Cooperative Fellowship and in part by NSF Grant CDA-9214908.

References

[1] T. M. Austin and G. S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs", *Proc. of 19th ISCA*, 1992.

[2] D. Callahan, S Carr, and K. Kennedy, "Improving Register Allocation for Subscripted Variables", *Proc. of PLDI*, 1990.

[3] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching", *Proc. of 4th ASPLOS*, 1991.

[4] S. Carr and K. Kennedy, "Scalar Replacement in the Presence of Conditional Control Flow", *Software--Practice and Experience*, vol. 24(1), Jan. 1994.

[5] T. Chen and J. Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches", *Proc. of 5th ASPLOS*, 1992.

[6] T. Chen and J. Baer, "Effective Hardware Based Data Prefetching for High-Performance Processors", *IEEE Transactions on Computers*, vol. 44, no. 5 (May 1995).

[7] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching", *Proc. of 25th MICRO*, 1991.

[8] C. Chi and H. Dietz, "Unified Management of Registers and Cache Using Liveness and Cache Bypass", *Proc. of PLDI*, 1989.

[9] T. Diep, "VMW: A Visualization-based Microarchitecture Workbench." Ph.D. Thesis. Carnegie Mellon University, June 1995.

[10] J.W.C. Fu and J. H. Patel, "Stride Directed Prefetching in Scalar Processors", *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992.

[11] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformation", *Journal of Parallel and Distributed Computing*, vol 5, 1988.

[12] G. H. Golub and C.F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, 1989.

[13] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies", *Proc. of ICS*, 1991.

[14] K. Grimsrud, et al, "On the Accuracy of Memory Reference Models", *Proc. of 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, May 1994.

[15] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor", *IBM Journal of Research and Development*, Vol 34, Num 1, January 1990, pp. 37-58.

[16] L. Gwennap, "Digital Leads the Pack with 21164", *Microprocessor Report*, Vol 8, Num 12, September 12, 1994.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

[18] F. Irigoin and R. Triolet, "Supernode Partitioning", *Proc. of 15th POPL*, 1988.

[19] A. C. Klaiber and H. M. Levy, "An Architecture for Software-Controlled Data Prefetching", *Proc. of 18th ISCA*, 1991.

[20] G. D. McNiven and E. S. Davidson, "Analysis of Memory Referencing Behavior For Design of Local Memories", *Proc. of 15th ISCA*, 1988.

[21] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", *Proc. of 5th ASPLOS*, 1992.

[22] V. Sarkar and G. R. Gao, "Optimization of Array Accesses by Collective Loop Transformations", *Proc. of ICS*, 1991.

[23] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools", *Proc. of PLDI*, 1994.

[24] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm", *Proc. of PLDI*, 1991.