Base Vectors: A Potential Technique for Micro-architectural Classification of Applications

Dan Doucette
School of Computing Science
Simon Fraser University
Email: ddoucett@cs.sfu.ca

Alexandra Fedorova
School of Computing Science
Simon Fraser University
Email: fedorova@cs.sfu.ca

***Abstract*—In this paper we examine the use of base vector applications as a tool for classifying an application's usage of a processor's resources. We define a series of base vector applications, simple applications designed to place directed stress on a single processor resource. By co-scheduling base vector applications with a target application on a CMT multi-threaded processor, we can determine an application's sensitivity to a particular processor resource, and an application's intensity with respect to a particular processor resource. An application's sensitivities and intensities for a set of processor resources comprise that applications sensitivity and intensity vectors. We envision that sensitivity and intensity vectors could be used for *(a)* understanding micro-architectural properties of the application; *(b)* forecasting an optimal co-schedule for an application on a multi-threaded processor; *(c)* evaluating the suitability of a particular architecture or micro-architecture for the workload without executing the workload on that architecture. We describe the methods for constructing base vector applications and for obtaining an application's sensitivity and intensity vectors. Using UltraSPARC T1 *(Niagara)* as the experimental platform, we validate base vectors as a method for classifying applications by showing that sensitivity and intensity vectors can be used to successfully predict the optimal static co-runner for an application.**

I. Introduction

For a given processor architecture, an application's use of key resources such as level 1 *(L1)* instruction and data cache, level 2 *(L2)* unified cache, floating point units and branch predictors will determine in large extent how well a given application will perform on a specified processor, and how well an application will run when co-scheduled with other applications on a multi-threaded *(MT)* processor. To a varying extent, each target application will stress a processor across the above resources differently. Understanding how an application places stress on each processor resource is key to discovering how optimal co-schedules can be constructed and how different processor architectures may be more optimal for a given application set. In this paper we explore the use of base vector applications to classify target applications with respect to their resource use and to use that classification to forecast relative performance in different co-schedules. Base vector applications, or BV applications, are small applications designed to both *(a)* place direct stress on a single processor resource and *(b)* be significantly dependent on that processor resource. *(Resource dependence means that the BV application's performance will significantly degrade if it is deprived of that resource.)* By co-scheduling the BV applications with target applications and measuring the relative slowdowns of each application, we determine an applications sensitivity and intensity with respect to a particular processor resource.

Within the context of this paper, *sensitivity* refers to how an application's performance responds to a lack of a particular processor resource. By co-scheduling a target application with a BV application that stresses a particular resource and measuring the target application's slowdown *(relative to when it runs on its own)* we determine the target application's sensitivity to that resource. For example, if a target application co-scheduled with a BV application stressing resource X experiences a greater slowdown then when co-scheduled with a BV application stressing resource Y, we say that the target application is more sensitive to resource X than to resource Y.

Intensity refers to how much an application stresses a particular processor resource. Recall that BV applications are designed to be dependent on the resource they target, so by co-scheduling a particular BV application with target applications and measuring the slowdown of the BV application *(relative to when it runs on its own)*, we determine the target application's relative intensity to the resource targeted by the BV application. For example, if the BV application targeting resource X experiences greater slowdown with application *A* than with application *B*, we say that application *A* is more intensive with respect to resource *X* than application *B*. An application's sensitivity and intensity vectors can be used for understanding the micro-architectural properties of the application and for analyzing how an application's performance will change depending on the co-schedule on a multi-threaded processor, or depending on the architecture it runs on. Some concrete examples are:

Relative performance in different co-schedules: Given a target application *T* sensitive to a resource *Y*, and two applications *A* and *B*, such that *A* is more intensive with respect to *Y* than *B*, we can project that *T* will run more slowly in co-schedule *(T,A)* than in co-schedule *(T,B)*.

Cross-microarchitecture performance forecast: Given a target application *T* sensitive to a resource *Y*, we can project that *T* will perform worse on microarchitecture *A* than on microarchitecture *B* if *A* has less of the resource to which *Y* is sensitive to than *B*. The amount of resources on the two micro-architectures can be measured using BV applications,

by comparing the relative performance of each BV applications on the two micro-architectures. Cross-microarchitectural studies are only outlined here and will be studied in future work.

In this paper we validate the effectiveness of BV technique to expose applications microarchitectural resource sensitivities and intensities by using it to forecast relative performance in co-schedules. For our experiments we use the Sun Microsystems UltraSPARC T1 system with eight four-way multi-threaded cores [6]. We find in general that we can correctly predict the most optimal co-runner in 4 out of 6 case studies that we carried out.

The remainder of this paper is organized as follows. Section 2 describes the construction of the BV applications. Section 3 describes the methods we used to gather sensitivity and intensity data from base vectors and target applications. Section 4 provides the detailed results of intensity and sensitivity measurements and the base vector co-scheduling experiments. Section 5 discusses related work, and Section 6 outlines future work and summarizes.

II. BV Applications

BV applications are very small programs designed to place stress on a single processor resource. By placing stress on a single processor resource and measuring the effect of this stress on the run times of other target applications, the relative usage of this processor resource by the target application can be determined in a relative manner. For the UltraSPARC T1 processor, we created 5 base vectors for each of the following processor resources:

- L1 Instruction Cache (I-Cache-L1)
- L1 Data Cache (D-Cache-L1)
- L2 Cache via L1 I-Cache (I-Cache-L2)
- L2 Cache via L1 D-Cache (D-Cache-L2)
- Floating Point Unit (FPU)

We have not created a BV application for the branch predictor, because branch prediction is not done on the UltraSPARC T1 [12].

A. Instruction Cache Base Vectors

The instruction cache base vectors are simple applications designed to stress the instruction cache of the processor. The intent of the application is to create the worst-case usage of the instruction cache. For the UltraSPARC T1 platform, this is accomplished in assembly language using the unconditional branch instruction. By creating a loop of branch instructions the size of the instruction cache, an application can create intense pressure on the instruction cache while exhibiting little or no stress on other resources.

As outlined in 1, the instruction cache base vector consists of a series of unconditional branch instructions, evenly spaced by the size of a single cache line. When the processor executes these instructions, a high degree of stress will be placed on the instruction cache because only 2 instructions per cache line will be executed *(branch instruction and branch delay slot)* and the loop is the size of the entire cache. To complete

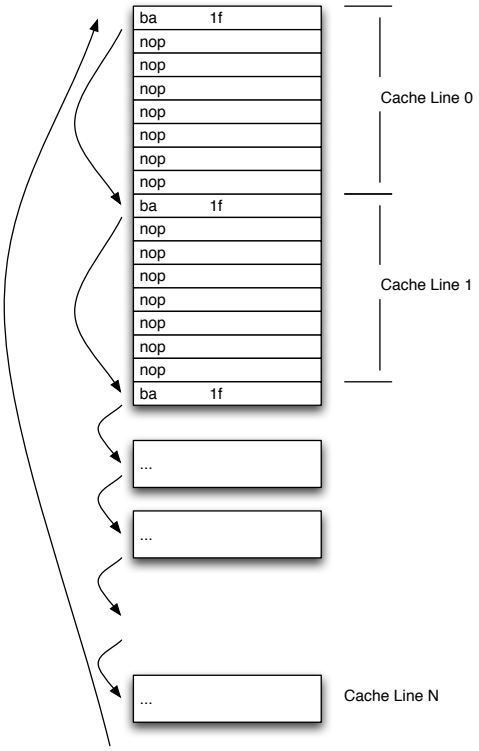

Fig. 1. I-Cache Base Vector Application

the construction of the base vector, the assembly code snippet outlined in figure 1 is cut-and-paste repeated to extend the size of this simple application to equal the size of the instruction cache. This code will be executed in a loop a fixed number of times to construct the overall base vector application. The number of loop iterations is fixed once at the time of creation to give a running time of the base application in the range of 30 to 60 seconds. Using the methods described above, both the ICache- L1 and I-Cache-L2 BV applications are created. The I-Cache-L1 BV application is sized to 16K bytes for the UltraSPARC T1 and the branch size is 32 bytes. The I-Cache-L2 BV application is sized to 3M bytes and the distance between branch instructions is 64 bytes.

B. Data Cache Base Vectors

The data cache base vectors are constructed using the concept of 'pointer-chasing'. A region of memory the size of the data cache is allocated and initialized to contain the address of the next cache line in memory. A simple piece of code will continually read the address of the next cache line in memory from the current line in memory. Repeating this operation causes the entire data cache to be accessed with minimal pressure on the instruction cache. An assembly code segment and associated data region are shown below.

As outlined in figure 2, the data cache base vector is created by initializing a data cache sized region of memory. The next cache line address is written at initialization time to the first word of each cache line. These addresses are loaded and used in immediate succession by the code highlighted in Figure 2 on

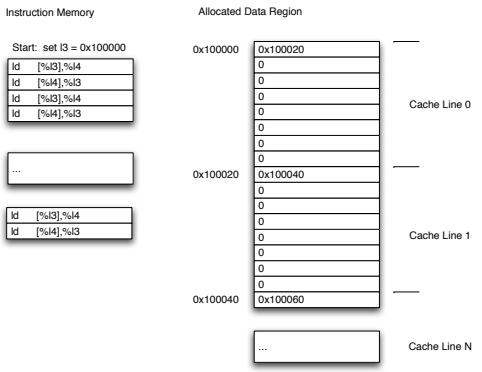

Fig. 2. D-Cache Base Vector Application

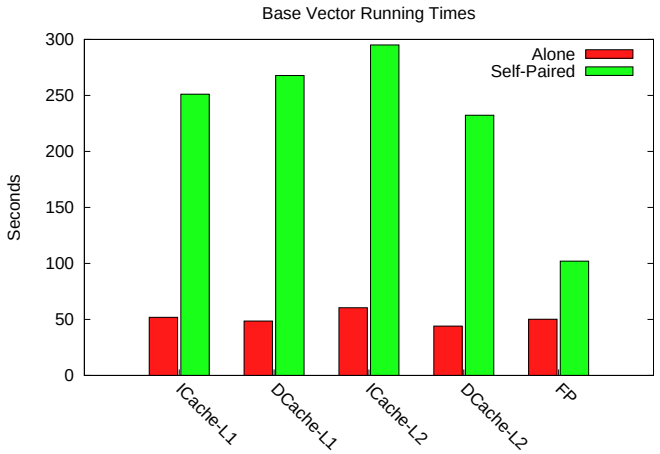

Fig. 3. BV Application run-times when running alone and co-scheduled with the identical BV application.

the left. This code essentially loads in a new cache line each instruction executed. The replicated load instructions shown are cut-and-paste repeated in the application several times to amortize any instructions required to loop back to the top of the code. The instruction size of this base vector however cannot grow too large, as it will begin to overly stress the instruction cache, so a balance is struck by keeping the loop size to within 10 instruction cache lines.

1) Operating system page sizes: For the instruction and data cache base vectors above, the general strategies require a large, contiguous region of memory to stress the cache. On the UltraSPARC T1 processor, the caches *(both L1 and L2)* are physically indexed. As both the instruction and data cache base vectors are created and executed as user programs, both applications will be arbitrarily translated into physical pages by the operating system using the processor's translation look-aside buffer *(TLB)*. Because of the arbitrary mapping by the TLB into physical address space, the instruction and data base vectors cannot be directly executed. For the UltraSPARC T1 platform, the Solaris operating system provides an API to allow the user application to control how memory is mapped into physical addresses. The *memcntl* API in Solaris allows the user to change the size of an arbitrary page mapping, allowing page sizes up to 256MB. For ICache- L1 base vector a 64KB physically contiguous page of memory was allocated and used. This size was sufficient to cover the entire 16KB L1 I-Cache on the UltraSPARC T1 processor. For D-Cache-L1 base vector, nothing special had to be done, because the default page size of 8KB was sufficient to cover the entire 8KB L1 D-Cache on the UltraSPARC T1. For ICache- L2 and D-Cache-L2 vectors, a 4MB physically contiguous page of memory was allocated and used. This size was sufficient to cover the entire 3MB level 2 cache on the UltraSPARC T1 processor. For the instruction cache base vector, the entire executable region was copied to this 4MB page and the base vector loop was executed directly from this region.

C. Floating Point Base Vectors

The floating point base vector application implemented for this paper was written in assembly language and simply executes a long string of floating point operations, one directly after the other. The long string of floating point operations is executed in a loop a fixed number of iterations to create an overall execution time ranging between 30 to 60 seconds. As with the data cache vectors, the unrolled string of floating point operations cannot be made too large for fear of creating too much stress on the instruction cache, thus for this paper the floating point base vector was kept to within 10 cache lines.

D. Validation of BV Applications

We present measurements showing that BV applications meet their goals of *(a)* stressing a particular resource and *(b)* being dependent on a particular resource. Figure 3 shows the run times of each of the five BV applications when they are run alone and when they are co-scheduled with the identical BV application. The data shows that in co-scheduled runs each BV application experiences significant relative slowdown. This demonstrates *(a)* that the co-scheduled BV-application stresses the target resource, and *(b)* the BV-application depends on that target resource.

III. Methods of Measurement

In order to determine a target application's sensitivity toward a specific processor resource, we co-schedule a specific base vector application with the target application. By co-scheduling each individual base vector with each application, we can create a histogram which outlines the application's sensitivity toward a specific processor resource. In turn, we can also measure the slowdown of the base application itself to determine how intensely a target application uses a specified processor resource. For target applications, we chose the SPEC CPU2000 suite of applications. We used 21 of the 26 available applications, excluding the FORTRAN benchmark applications due to compile issues with the UltraSPARC T1/GNU toolchain.

For this paper, the experiments outlined below were undertaken on the UltraSPARC T1 processor. The UltraSPARC T1 processor features 8 independent CPU cores with 4 threads

per core. In order to measure the effect of co-scheduling base vector applications and target benchmark applications, processes were set to execute on the same core. The data collected for this paper can be summarized into three distinct groups of results recorded.

A. Target Application Sensitivity

In order to determine a target application's sensitivity to a given processor resource, we co-schedule each processor resource BV application with the target application. We can then measure the percent slowdown of the target application with each base vector application by comparing the absolute time to complete the target application versus the time to complete the application running alone. The result is a single normalized slowdown measure for each BV application for a given target application. We measure the relative slowdown for six of the SPEC CPU2000 benchmark applications across each of the four BV applications targeting the caches *(I-Cache-L1, I-Cache- L2, D-Cache-L1, and D-Cache L2)*, producing 24 separate measurements in total. We limit the sensitivity analysis to six applications: *art, gzip, crafty, vortex, parser and twolf*. We were unable to run all combinations of BV-applications and target applications *(88 runs in total, requiring close to 88 hours)* due to limited availability of the test hardware. Nevertheless, our data allows us to gain insight into the effectiveness of the base vector technique. In the future, we will extend our sensitivity analysis for the entire SPEC CPU2000 suite.

B. Target Application Intensity

To determine how intensely a given target application uses a specified processor resource, we co-schedule the target application with each of the four BV applications. To determine intensity, we measure the effect of the target application on the base vector application by comparing the absolute run time of the BV application when co-scheduled with the target application against a run of the BV application running alone. We provide results for each base vector against each of the 21 SPEC CPU2000 benchmarks outlined above for 84 reported results in total. Due to the short relative run times of the base vector applications *(approximately 1 minute)* versus the longer SPEC benchmark run times, we were able to report results for all base vectors against all SPEC benchmarks outlined above.

C. Relative Performance in Co-Schedules

To determine how each SPEC benchmark performs when co-scheduled with each other, we use the six SPEC benchmarks identified above and co-schedule them with each other in pairs, measuring the relative slowdown of each benchmark. Given the measured performance effects of co-scheduling the benchmark applications, we would like to determine whether sensitivity and intensity measures for a given pair of target applications can be used to determine the relative slowdowns when the benchmark applications are co-scheduled. To gather the real benchmark co-scheduling information, the benchmark applications are co-scheduled on the same core as outlined above. Each benchmark application is run in a series of three iterations, with the results of the second iteration taken to ensure a complete overlapped execution of the two benchmark applications.

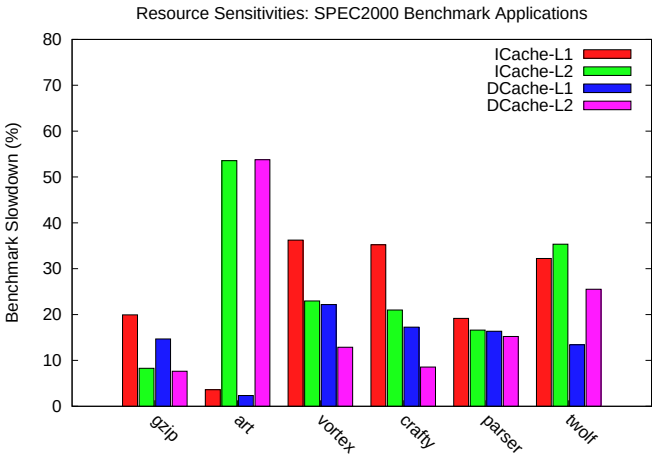

Fig. 4. SPEC benchmark application sensitivities to cache resources.

IV. Results

As outlined in the previous section, the methods for data collection and the results can be categorized into three groups, each of which is outlined below.

A. SPEC Benchmark Sensitivities

Outlined in Figure 4 are the measurements of the run times for the various SPEC benchmark applications when co-scheduled with each of the BV applications. *(Although not explicitly tabulated in this paper, it should be noted that the variances of the run times gathered for each co-schedule run were very low, certainly within 1 or 2 percent of the total run times tabulated.)* The histogram in this figure outlines the percent slowdown of the SPEC benchmark application when co-scheduled with each BV application. Slowdown percentages are calculated as a percentage of the time for the benchmark application to run alone, thus a 100% slowdown would indicate an absolute co-schedule runtime twice that of the time for the benchmark application to run alone. A large increase in benchmark runtime when co-scheduled with a BV application targeting resource *Y* indicates the target application's relative sensitivity to resource *Y*.

Observing figure 4, the range of sensitivities of the various benchmark applications can be seen. The *art* SPEC benchmark appears extremely sensitive to the L2 cache. This result is similar to one reported in previous work, which showed that *art* is very sensitive to the L2 cache when the processor cache size is between 1MB and 1.25MB [9]. That same work also reported that *twolf* is more sensitive to the L2 cache than *crafty*, *gzip* and *parser* for cache sizes larger than 512KB *(no results reported for vortex)*. Our results similarly show that *twolf* is more sensitive to the L2 cache compared to *crafty*, *gzip* and *parser*.

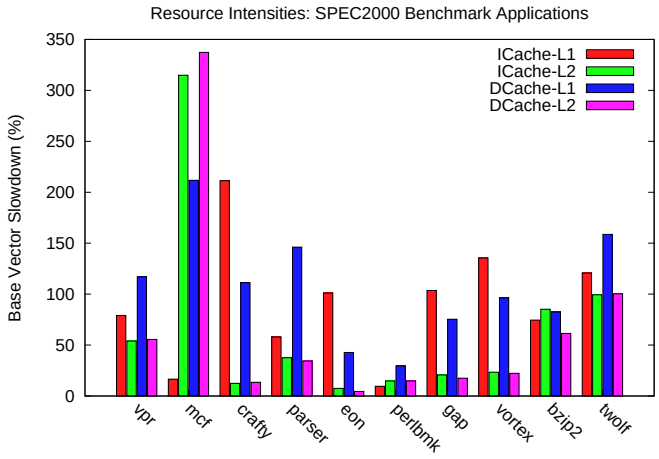

Fig. 5. SPEC benchmark application intensities (I).

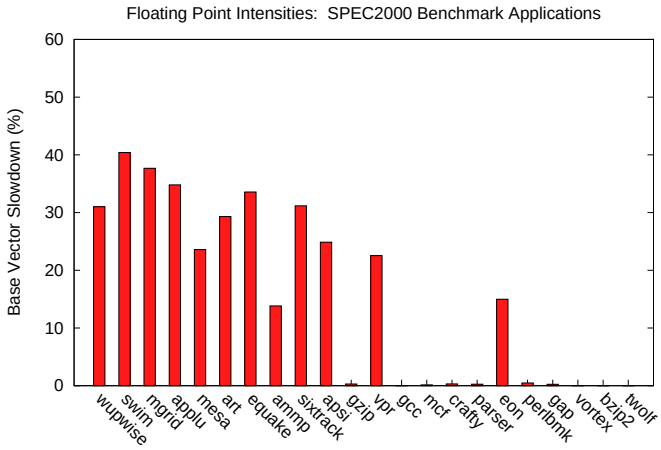

Fig. 7. SPEC benchmark application FPU intensities.

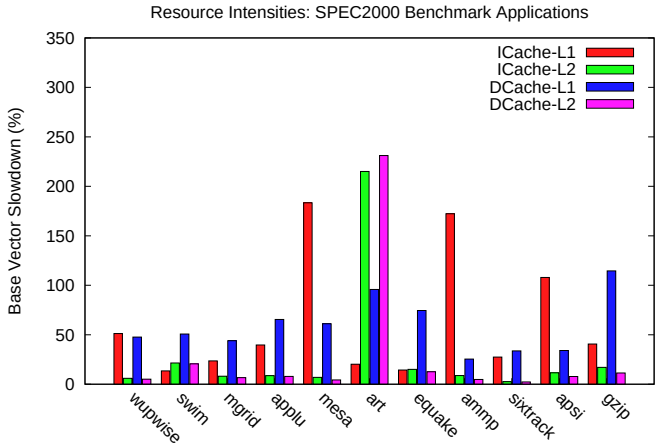

Fig. 6. SPEC benchmark application intensities (II).

B. SPEC Benchmark Intensities

Outlined in figures 5 and 6 are the percent slowdown measurements for the BV applications when run alone vs. when co-scheduled with each SPEC benchmark application.

As each BV application is designed to be dependent on a single processor resource, the slowdown observed in its runtime when co-scheduled with a benchmark application can be attributed entirely to the amount stress that the benchmark application puts on the BV applications target resource. As such, these figures show the benchmark application's relative intensity toward each processor resource. As shown in figures 5 and 6, the changes in the base vector application's run times are very pronounced, with the largest (over 300%) slowdown seen by the L2 D-Cache base vector application when co-scheduled with the *mcf* SPEC benchmark. Another application with high L2 cache intensity is *art*. These results confirm previously reported high L2 cache intensities for *mcf* and *art* [1,4,5].

C. Floating Point BV Application Usage

We present the measurements showing benchmark applications intensity with respect to the floating point unit. *(We did not measure benchmark applications sensitivities to the FPU due to time constraints.)* To gain a measure for intensity, the floating point BV application was co-scheduled with each of the SPEC benchmark applications, noting the absolute run times of the base vector when co-scheduled with each of the benchmark applications. The results are shown in Figure 7. As shown in figure 7, the intensities of each of the floating point SPEC CPU benchmarks *(shown on the left)* vary to a large extent. The SPEC integer applications show zero intensity, save two. The integer benchmarks *eon* and *vpr* show considerable floating point intensity. (The FPU of integer benchmark *vpr* exceeds that of floating point benchmark *ammp*!) The presence of floating point instructions was confirmed by inspecting the source code for both SPEC benchmark applications. The ability of the BV applications to detect this floating point activity without prior knowledge about the application in some respect affirms the base vector approach to application classification.

D. SPEC Benchmark Co-Scheduling Results

Given a target application *A* co-scheduled with applications *B* and *C*, we wanted to determine if *A's* relative slowdown in co-schedules *(A, B)* and *(A, C)* can be determined using the sensitivity vector of *A* and the intensity vectors of *B* and *C*. Intuitively, if *A* is sensitive to a resource *Y*, it will experience a larger slowdown with *B* than with *C* if *B* is more intensive with respect to resource *Y*. We use intensity and sensitivity vectors of co-scheduled applications to predict relative slowdowns in different co-schedules. In our experiments, an application's sensitivity vector is the set of slowdowns *(in percent)* for that application when co-scheduled with the four BV applications targeting the cache resources. An application's intensity vector is the set of slowdowns *(in percent)* for the four BV applications when they are co-scheduled with the target application. We will refer to these

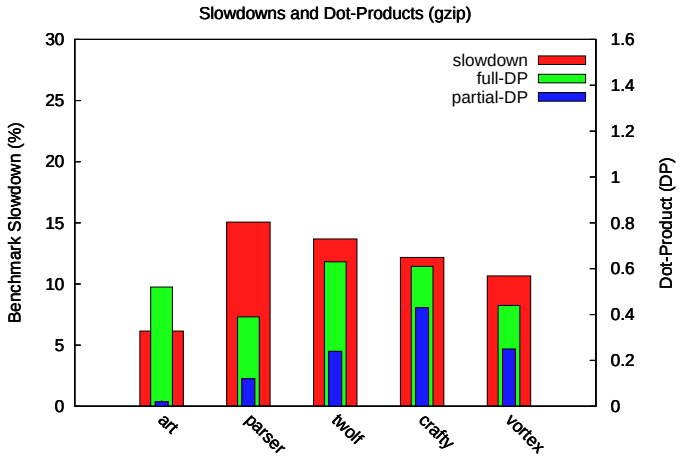

Fig. 8. 'gzip' slowdowns and dot-products

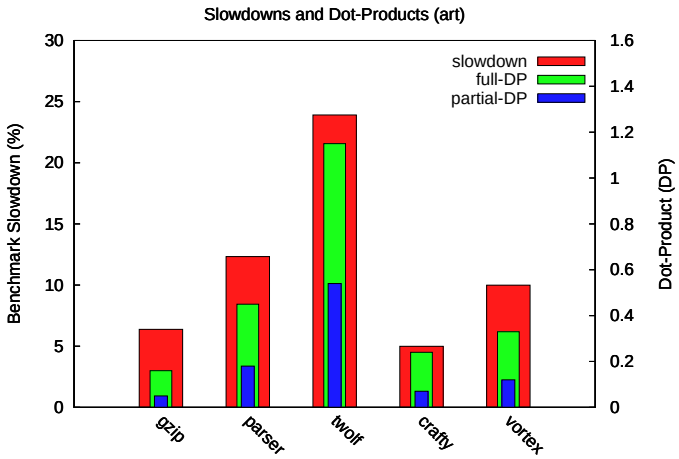

Fig. 9. 'art' slowdowns and dot-products

two vectors for a hypothetical application T as:

$$\vec{V_{sens}} = (A, B, C, D) \quad (1)$$

$$\vec{V_{intens}} = (W, X, Y, Z) \quad (2)$$

where *A*,*B*,*C*,*D* represent the target application's slowdowns when run with each of the four BV applications and *W*,*X*,*Y*,*Z* represent each of the four BV application's slowdowns when run with the target application.

Assuming that we want to forecast *A's* relative slowdown in co-schedules *(A, B)* and *(A, C)*, we take the two dot products:

$$DP_{AB} = \vec{V_{sens}}(A) \bullet \vec{V_{intens}}(B) \quad (3)$$

$$DP_{AC} = \vec{V_{sens}}(A) \bullet \vec{V_{intens}}(C) \quad (4)$$

Comparing the two dot products, we expect that a higher value of the dot product will indicate a larger predicted slowdown for *A* in the corresponding co-schedule.

We experiment with two techniques: *(1)* **Full dot products:** we use the dot products of the entire sensitivity and intensity vectors to make slowdown estimates. *(2)* **Partial dot products:** we use the target application's most sensitive resource only. This amounts to using only a single dimension of the sensitivity vector outlined above.

For each of our chosen six SPEC benchmarks we present the relative slowdown for each benchmark when co-scheduled with each other benchmark and the full and partial dot-products for each SPEC benchmark application. All measurements are shown in figures 8 through 13.

Examining the results for *art* in figure 9, we see a clear visual relationship between the calculated dot-products and the relative slowdowns of *art* when co-scheduled with the other SPEC benchmarks. For the other benchmarks the results are not as clear, but do exhibit some similarity to the measured slowdowns.

The results for *twolf* also exhibit a clear relationship between the calculated dot-products and the measured slowdowns for each co-schedule. The only exception in the *twolf*

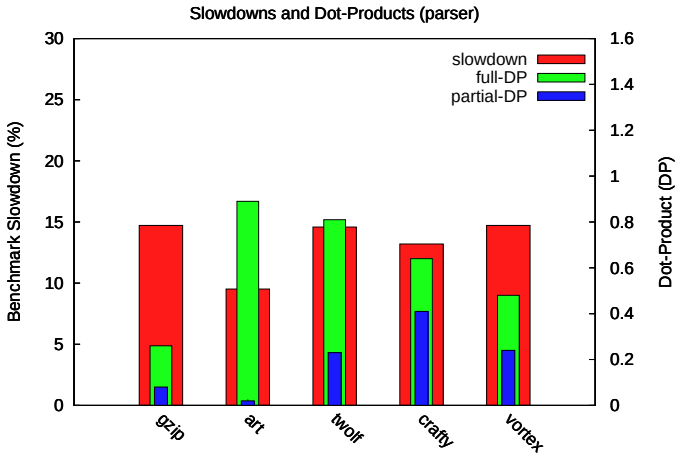

Fig. 10. 'parser' slowdowns and dot-products

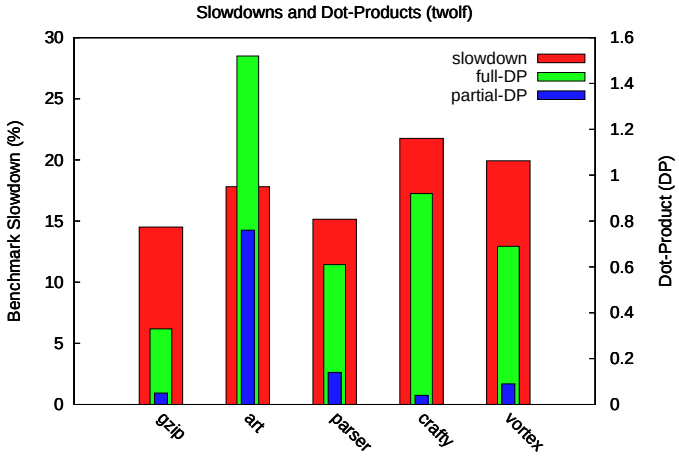

Fig. 11. 'twolf' slowdowns and dot-products

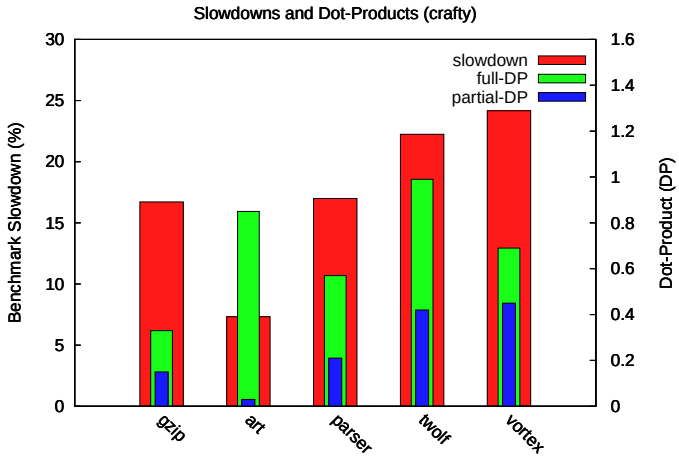

Fig. 12. 'crafty' slowdowns and dot-products

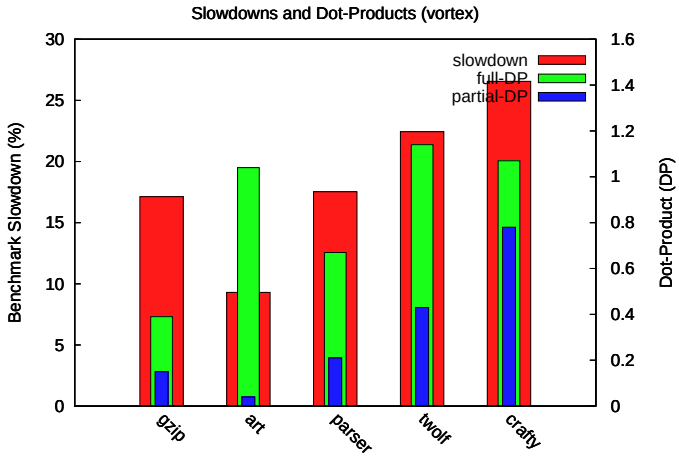

Fig. 13. 'vortex' slowdowns and dot-products

results is the dot-product prediction of *art* as the worst co-runner. The over-pessimistic prediction of *art* was also made for *parser*, *crafty* and *vortex* (figures 10, 12, 13). The very large I-Cache-L2 and D-Cache-L2 intensity measurements for *art* account for these over estimates. Looking at the partial dot-products for *parser*, *crafty*, and *vortex*, the over-pessimistic results are replaced with more accurate predictions of the worst co-runners as *parser*, *crafty* and *vortex* are all most sensitive to the L1 I-Cache and thus do not use *art*'s very large intensity values in their partial dot-product calculations.

Examining the results for *crafty* and *vortex* further, we see the partial dot-products very accurately depict the relative slowdowns of the target application when co-scheduled with each of the SPEC benchmarks. The sensitivity profiles for these two applications are nearly identical, with a well defined most sensitive resource *(L1 I-Cache)*. The partial dot-products correctly forecast that the worst slowdowns will occur when those applications are run with each other. The partial dot products also correctly forecast that *art* will be the best co-runner for these applications. We now refer to Figure 10, showing the slowdowns and dot-products for *parser*. Referring to figure 4, *parser* does not show well defined sensitivities for any particular resource, and so it is inherently difficult to estimate the best co-schedule. Nevertheless, the partial dot product correctly predicted the optimal co-schedule with *art*.

E. Discussion

In the previous section, we have seen the partial dot-product correctly predict the most optimal static co-runner for four of the six SPEC benchmarks identified. Also, outlined in figures 8 through 13, we have visually presented the relationship between the measured slowdowns of each of the benchmark co-schedules and the dot-products calculated using sensitivity and intensity vectors. Although this technique has shown some amount of success in choosing an optimal co-schedule, the results do not show this to be a clearly accurate technique for the creation of task schedules.

As mentioned in the introduction, the base vector technique is not ideally suited for a scheduler application, but rather may be used to simply gain insight into the behavior of an application. While the results for optimal co-schedules are less than perfect, the relationships between the dot-products calculated and measured slowdowns are quite pronounced. In order to improve the accuracy of the information provided by BV applications we intend to expand the technique to; *(a)* create independent L2 BV applications which currently are dependent, *(b)* measure the range of cache sensitivities by varying the size of the BV applications.

V. Related Work

Similar work mostly relates to predicting application slowdowns in different co-schedules on multi-threaded processors. Snavely et al. used a heuristical approach to estimate which schedules will show a better symbiosis with respect to resource sharing [10]. They used hardware counter measurements of different resources as inputs into their heuristical model. They have not, however, developed a formal classification scheme with respect to applications resource intensity and sensitivity. Another body of work has to do with development of explicit analytical models for estimating slowdowns in different co-schedules [1,2,3,7,11]. The models are more complex than the base vector technique, largely microarchitecture dependent, and do not provide a way to classify the applications sensitivity and intensity. One recent study that is perhaps the most similar to ours is on Micro-architectural Scheduling Assist (MASA) [8]. MASA is a tool that co-schedules applications on a hyper-threaded processor based on their resource use (a concept similar to our resource intensity). That study, however, does not provide a formal well-defined technique to measure intensities and does not take into account the sensitivities.

VI. Summary and Future Work

In this paper we introduced the concept of a base vector and outlined a technique to use the BV applications for microarchitectural classification of an application's resource use. In an attempt to validate to some extent the ability of the BV applications to act as an application classifier, we

applied their use to the task of choosing optimal co-schedules. Using an application's most sensitive resource as input into the predictive model, we find that 4 out of 6 optimal static co-runners were predicted correctly. The BV applications ability to find the most intensive and sensitive resources used by an application may be considered partially validated. Although used for co-scheduling in this paper, the BV applications and methods outlined are most likely better suited for alternate tasks. Researchers and other CPU power users could employ the classification abilities of the BV applications to peer into target applications. By understanding how the scientific workload uses processor resources, and understanding the workload's sensitivities to processor resources, better choices could be made when identifying hardware resources for projects and workload optimization efforts could be focused to reduce application sensitivities that are identified. Base vector applications could be used in embedded systems to identify application resource intensities and sensitivities for environments which may not be conducive toward other types of information retrieval. Hardware counters on such systems may be lacking in numbers and may be difficult to interface to. Simulations of such systems tend to be rudimentary and may not allow gathering of detailed information regarding cache access and other microarchitectural resource usage. Furthermore, hardware counters alone may not provide enough detailed information about the embedded workload to enable hardware and software designers to make intelligent decisions regarding the next generation of device. Using BV applications, the embedded workload's most sensitive resources can be easily found, giving direct and meaningful input into which areas of the hardware to improve in the next generation.

VII. Acknowledgments

The authors would like to thank Sun Microsystems for providing support for this work.

References

[1] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Multi-Processor Architecture. In Proceedings of the 12th International Symposium on High Performance Computer Architecture, pp. 340-351, 2005

[2] A. El-Moursy, R. Garg, David Albonesi, and Sandhya Dwarkadas. Compatible phase co-scheduling on a CMP of multi-threaded processors. In Proceedings of the 20th International Parallel and Distributed Processing Symposium, 2006

[3] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. A Non-Work-Conserving Operating System Scheduler for SMT Processors. In Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA-33, pp. 10-17, 2006

[4] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Cache-Fair Thread Scheduling for Chip Multiprocessors. Harvard University Technical Report TR-17-06, 2006

[5] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 111-122, 2004

[6] Poonacha Kongetira. A 32-way Multithreaded SPARC(R) Processor. In Proceedings of the 16th Symposium On High Performance Chips (HOTCHIPS), 2004

[7] Tipp Moseley, Joshua L. Kihm, Daniel A. Connors, and Dirk Grunwald. Methods for Modeling Resource Contention on Simultaneous Multithreading Processors. In Proceedings of the International Conference on Computer Design, pp. 373-380, 2005

[8] Jun Nakajima and Venkatesh Pallipadi. Enhancements for Hyper-Threading Technology in the Operating System - Seeking the Optimal Scheduling. In Proceedings of the Second Workshop on Industrial Experiences with Systems and Software, 2002

[9] M. K. Qureshi and Yale Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In Proceedings of the 39th International Symposium on Microarchitecture, 2006

[10] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 234- 244, 2000

[11] G. E. Suh, S Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In Proceedings of the 8th International Symposium on High Performance Computer Architecture, pp. 117-128, 2002

[12] Sun Microsystems. OpenSPARC T1 Microarchitecture Specification. http://opensparct1.sunsource.net/specs/OpenSPARCT1_Micro_Arch.pdf, 2007