Analysis of the Execution of a Next Generation Application on Superscalar and Grid Processors

Juan Rubio and Lizy Kurian John
Electrical and Computer Engineering
The University of Texas at Austin
{jrubio,ljohn}@ece.utexas.edu

Abstract

The astonishing advances in processing speeds and the phenomenal increase in chip densities have enabled the creation of very powerful microprocessors and computer systems. Future high end computing systems are expected to have Teraflops of computing capability and massive amounts of storage. Such computers are expected to be important for discovery in the fundamental sciences, pharmaceuticals, and several other causes for the improvement of mankind.

In this paper, we analyze a workload that is expected to be instrumental in designing and architecting future computing systems. The workload is a ground motion tracker indication (GMTI) application created by the scientists at the MIT Lincoln Laboratory. The application is being used to drive the design of several advanced future computer systems; hence it is important to understand the computational, memory access and parallelism features of this application. In this paper, we first describe the various components/stages of this application. Then, we perform detailed analysis of the execution of this application. On the basis of profiling the execution of the application, both on actual platforms and with simulations, we show that the parallelism in the several stages of the application is different. The application is seen to contain a large amount of parallelism that can be exploited by spatially or temporally parallel computer architectures. However, the non-uniformities in computing requirements as well as memory access patterns of the different stages are important considerations in the design of spatially/temporally parallel architectures to handle these applications. The execution of the application on a superscalar processor and a grid processor are analyzed.

1 Introduction

Computer architects and designers often wish to design microprocessors and computer systems that cater to emerging and future workloads. “What's the next killer application?”, is a question frequently asked by computer designers. In a panel presentation last year on future workloads [1], a panelist held the notion that future applications are going to be related to “Life, Death or Games”. ‘Life’ stood for life sciences, pharmaceuticals/drug discovery, biological and microbiological research, etc. ‘Death’ referred to military applications, weapon simulations, crash analysis, and radar and sonar processing, among others. And in the panelist's opinion, interactive and multiplayer games, games needing natural language recognition and semantic analysis are going to be dominant future workloads as well.

Structured or regular computing seems to be a feature of the ‘life’ and ‘death’ categories. Applications in the physical, chemical and biological sciences are typically regular and structured. Perhaps the ‘games’ category contains some processing similar to SPECint [2], however part of ‘games’ is also media processing, which does have a regular component. It is interesting to note that several of these applications are more similar to scientific and technical workloads exemplified by SPECfp benchmarks [2] or STREAM benchmarks [3] than to popular benchmarks used by most architects and designers.

In recent years, scientific and technical workloads have fallen into some disrepute. However, looking at the prediction from this panelist, they seem to be at the heart of the processing required in future computing systems. Hence, we decided to look at a workload that scientists at MIT Lincoln Laboratory have created to drive the Polymorphous Computing Architecture (PCA) initiative [4] of DARPA. It is an Integrated Radar-Tracker (IRT) application [5, 6, 7] that consists of a Moving Target Indicator (MTI) and a Kinematic Tracker (KT). The function of the MTI is to process the data generated by the radar antenna and detect moving objects in the controlled space. The KT component keeps track of the location of the objects and their trajectories during a period of time. This workload shows the need of computing that will not be satisfied by conventional uniprocessors or superscalar processors. Parallel processing systems performing several computations at the same time are required to satisfy the computational requirements of these types of workloads.

In this paper we look in detail at the different components of the Moving Target Indicator application. First, we describe the various stages of the application. Then we present the computation requirements of each stage. We analyze the parallelism in the application and the issues in exploiting pipelining and parallelism. Finally, we analyze the performance of this application on a superscalar processor and an emerging grid processor architecture.

2 Radar Tracker Application

2.1 Radar Operation

Radar systems are used in myriad situations. They are used to identify the different types of soil in a region, to create a map of the topography of a terrain, to detect airplanes, etc. The function of the radar system we study is to detect moving objects in a controlled volume. To detect the objects, the radar unit activates its transmitter and sends out a short high-frequency radio pulse. The radar unit then turns off the transmitter module, activates the receiver and listens for an echo. The system estimates the distance of the object based on the time it takes for the echo to arrive. It can also estimate the speed of the object based on a shift of the frequency (doppler effect) [8] and previous kinematic information about the object.

Radars can be further classified depending on the type of objects they study. Air-based radars are used to track the movement of air-borne objects, while ground-based radars track the movement of objects on the ground. For ground-based radar there are more potential interferences than in air-based radar, since the radar pulses echo from the objects being tracked as well as from the ground and other stationary objects. To remove these interferences, the system needs to consider only those *returns* (objects) that are *Doppler-shifted.*

2.2 Radar Processing Components

The main components of the IRT application studied for this work are a Ground Moving Target Indicator (GMTI) [9] and a Kinematic Tracker (KT). The GMTI module is a ground-based MTI. It processes the echo data from the radar and outputs information about the location, speed and trajectory of the detected targets. The KT gets this information and keeps track of the detected objects across time (which ones are new, which ones are old, and how they are moving). The IRT system is also assisted by the High Range Resolution (HRR) module. When the GMTI detects a target, the radar sends a directed waveform at the target to obtain additional information about it. This information is processed by the HRR module and it is needed by the KT to differentiate targets when their trajectories cross over. Figure 1 shows the block diagram for a simple Integrated Radar Tracker (IRT) system [6].

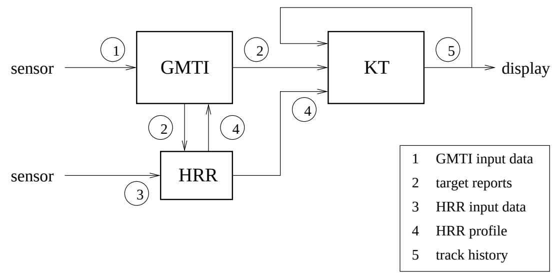

Figure 1: Block diagram of the IRT.

2.3 Ground Moving Target Indicator

The GMTI is a module that can identify returns corresponding to moving targets from those that correspond to stationary objects (or *clutter*) [8]. The most common technique is Doppler speed, and consists in detecting the returns that come with an associated shift in the wave frequency. Figure 2 shows the basic modules of the GMTI application we use in this work.

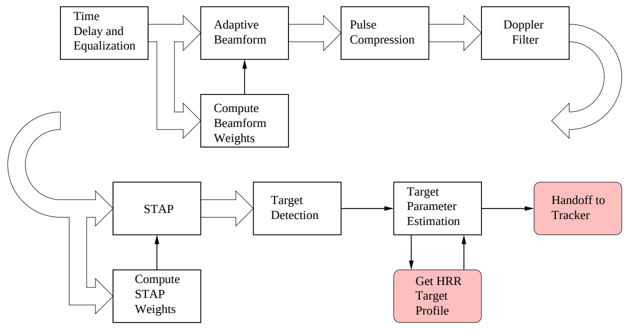

Figure 2: Block diagram of the GMTI. The shaded boxes represent modules outside the GMTI.

The input data to the GMTI application is the *data cube* generated by the radar sensors. A data cube is a three-dimensional array of complex numbers with dimensions *channels*, *range* and *pulses* [9]. Figure 3 shows the data cube and the way it is accessed by each of the components of the GMTI. The shaded regions

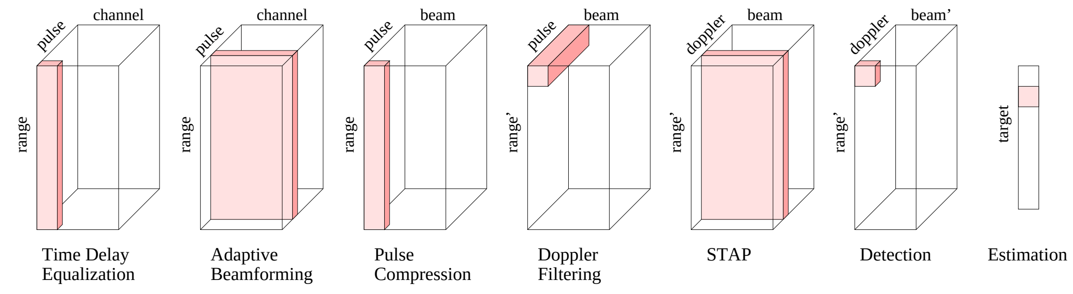

Figure 3: Data used by each stage of the GMTI.

correspond to elements of the array that are accessed at once to process that portion of the data cube.

- Time Delay and Equalization (tde)
  The TDE stage is applied to each data cube and it uses a finite impulse response (FIR) filter [10] on each range vector for each channel and pulse to compensate for signal differences between channel sensors.

- Adaptive Beamforming (abf)
  The ABF stage is applied to the resulting data cube. It transforms the filtered data into the beam-space domain to allow the detection of target signals coming from a particular set of directions of interest while filtering out spatially-localized interference (*jamming*). This is done by computing the weight matrix (*beamform matrix*) and applying this matrix over all the PRI matrices of the input data cube.

- Pulse Compression (pc)
  Pulse Compression uses a finite impulse response (FIR) filter on each pulse and channel to concentrate the signal energy of a relatively long transmitted radar pulse into a shorter pulse response.

- Doppler
  The doppler filter processes the data so that radial velocities of the target relative to the platform can be determined. Doppler filtering accomplishes this using a FFT applied to pulse data of each range gate and beam. To smooth out the ringing effect in the FFT, a temporal windowing is done prior to FFT using a Chebyshev window filter [10]. A further step which aids the STAP stage in filtering out ground clutter involves taking staggered pulse sample sets.

- Space-Time Adaptive Processing (stap)
  The STAP stage is a second beamforming stage which removes further spatially-localized radar interference and ground (space and time distributed) interferences. This is done by computing the STAP filter matrix (weight matrix) for each doppler bank and applying this matrix over the whole input data cube.

- Detection
  The target detection is performed on the processed radar data cube to produce a data structure which contains the co-ordinates of the detected targets. Target detection is done using CFAR (Constant False Alarm Report) detection to determine whether a target is present and then uses target grouping to eliminate multiple target reports where only one target is present. To perform the CFAR detection, the squared power of each cell in the data cube is computed. A local noise estimate is then computed for the cells under test. The power is then normalized and if it exceeds a noise threshold, the cell is considered to have a target.

- Estimation
  This component estimates the location of the targets in terms of the azimuth, distance and radial velocity. It also requests a set of high range resolution (HRR) measurements for each of the targets.

2.4 Memory requirements

Figure 3 introduces the notion of a data cube. The dimensions of the data cube depend on the parameters of the radar system. For our configuration, we use: 31 pulse repetition intervals, 9 channels, and 2691 range gates. Each element of this data cube is a complex number represented as 2 single-precision floating point values. Thus the input data cube for the GMTI is 5.72 MB.

Figure 3 also shows the way in which data is accessed by each of the stages. The highlighted block represents the data that is read in a single iteration of the loop. So for example, the *tde* stage accesses all the range elements for a given channel and pulse before it produces the corresponding output vector. This information tells us about:

- the order of traversal of the data structures. This can help us decide how to structure the arrays in memory to improve the spatial locality exploited by the caches or prefetching engines.

- the level of independence of the stages and how they can be parallelized. Furthermore, if two stages share a similar traversal mode (e.g. *tde* and *abf*), they could be fused, which has the potential of reducing the memory pressure.

3 Analysis of the stages

In the previous section, we described the functionality of the GMTI stages. In this section we look at some characteristics of their execution.

3.1 Base system

We perform our study in the context of the Grid Processor [11]. Grid is a polymorphic processor being developed as part of the TRIPS system [12]. Figure 4 shows a block diagram of a grid processor with 16 computation tiles, all with identical computation, storage and communication capabilities. Table 1 shows the configuration of the base grid system used in our experiments. The GMTI application is compiled using the Trimaran framework [13], and the resulting code scheduled for the grid processor using a custom block scheduler [12].

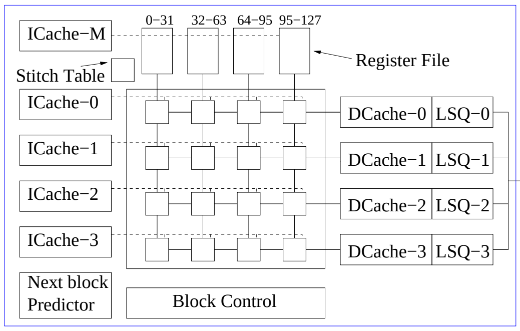

Figure 4: Block diagram of the grid processor. The instruction cache is located on the left side. Instructions are loaded into the computation tiles where they execute. Each tile has a 128-entries instruction buffer. Results move between the tiles using the on-chip interconnect.

We also compare these results with those obtained from conventional superscalar processors. We perform simulations of the GMTI benchmark using MASE [14], a simulation tool set derived from Simplescalar [15]. Superscalar processors with issue widths ranging from 4 to 16 were used in our experiments. Table 2 shows the configuration of the base superscalar configuration. For these experiments, we compile the GMTI application using the Alpha compiler with full optimizations.

Table 1: Configuration of the grid processor.

Processor	4 GHz, 4x4, D-morph
Core	128 physical frames 16 tiles, each can execute integer, and floating point instructions
L1-I L1-D L2	32 KB, 32 B block, 2-way, 2 cycles 64 KB, 64 B block, 2-way, 3 cycles 2 MB, 128 B block, 2-way, 13 cycles
Memory	50 ns

Table 2: Configuration of the base superscalar processor.

Processor	4 GHz, 4-way
Core	4 int ALU 1 int mult/div 4 fp ALU 1 fp mult/div 2 memory ports 128 ROB entries 64 LSQ entries
L1-I L1-D L2	64 KB, 32 B block, 2-way, 1 cycle 64 KB, 64 B block, 2-way, 1 cycle 2 MB, 128 B block, 2-way, 13 cycles
Memory	50 ns

3.2 Execution profile

Table 3 shows the duration of the GMTI module in both base systems. We observe that in the grid system, the early stages of the GMTI application (*tde*, *abf*, *pc*, *doppler* and *stap*) account for the majority of the execution time. In the superscalar system, the filtering stages (*tde* and *pc*) take the longest time, followed by the adaptive stages (*abf* and *stap*). We observe that the grid system is $1.5x$ times faster than the superscalar system with similar number of computatio resources (16-way). We also note that the *estimation* stage is $2.1x$ times faster in the superscalar system than on the grid system. The reasons for these differences will become clear as we examine the characteristics of this application.

The 16-way superscalar system mentioned in the previous comparison has as many computation units

Table 3: Duration of the GMTI stages in grid and superscalar processors. The results are presented in millions of cycles.

	grid	superscalar	
Stage	4x4	4-way	16-way
tde	52.77	181.24	110.13
abf	54.09	75.97	44.71
pc	40.79	147.09	88.29
doppler	19.66	57.63	39.00
stap	54.94	92.06	51.18
detection	7.20	18.75	15.08
estimation	5.20	3.13	2.50
Total	234.67	575.86	350.88
Total (seconds)	0.0587	0.144	0.0877

as the 4x4 grid system. However, wire delays prevent designers from building such a wide superscalar system [16]. Given the current fabrication technology, a more realistic approach is then to have a 4-way superscalar. In that case, the grid processor shows a speedup of $2.45x$ over the superscalar processor.

We analyze the execution of the GMTI application to identify the routines that dominate each of the stages. Figure 5 shows the relative time spent in each stage for the grid system. The routines that constitute the GMTI application are grouped as:

- *FFT init*: initialization code for the fast Fourier transform functions.
- *FFT* and *IFFT*: the direct and inverse fast Fourier transform –used often to implement a filter.
- *Compute weight*: obtains the coefficients for the adaptive filters after solving the Wiener-Hopf equation. This process uses QR-decomposition and LQ-decomposition of the input matrices.
- *Apply weight*: performs a matrix multiplication of the coefficients and the input data.
- *CFAR*: the constant false alarm rate algorithm is used in the *detection* stage. It computes an estimate of the noise for each cell in the input matrix and estimates the probability of a target being present there.
- *Core*: main function for each stage, which calls the other functions.
- *Others*: assisting functions (mostly memory management).

We observe that the execution of the GMTI application is mostly dominated by the FFT functions and matrix multiplication. The profiles of the *tde* and *pc* stages are very similar, the FFT functions dominating most of their execution time. The *abf* and *stap* stages are similar also, but being dominated by the matrix multiplication routines. Based on what we

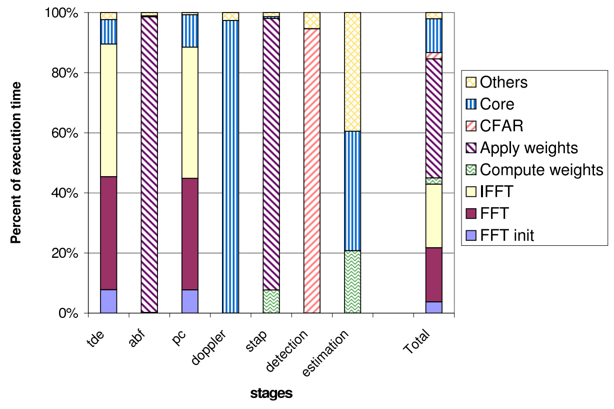

Figure 5: Fraction of execution time spent in major routines.

know about these algorithms [17], we expect a large amount of parallelism in the application. This profile gives us a glimpse to the advantages of having a large number of computational units for this application. As seen in Table 3, the wide superscalar processor (16-way) shows its largest speedups over the narrow superscalar processor (4-way) precisely in these stages.

3.3 Instruction Mix

Figure 6 shows the dynamic instructions mix observed in the grid system. The GMTI application performs computations over floating-point data, which accounts for 17% of the total instructions. However, most of the instructions executed correspond to integer arithmetic instructions. This is due to the use of complex addressing modes to access the 3-dimensional data cubes. The number of memory instructions is higher than the number of floating-point operations as many of the algorithms create arrays to store intermediate results. The early stages of the GMTI application are characterized for their regular loops. It is then easy for the compiler to unroll the loops which reduces the number of dynamic branches in the program. As seen in Figure 6, branches represent less than 5% of the total instructions in the first 5 stages. The last 2 stages have higher proportions of branches. The use of the CFAR algorithm in the *estimation* stage and the data dependent calculations in the *detection* stage contribute to their higher proportion of branches.

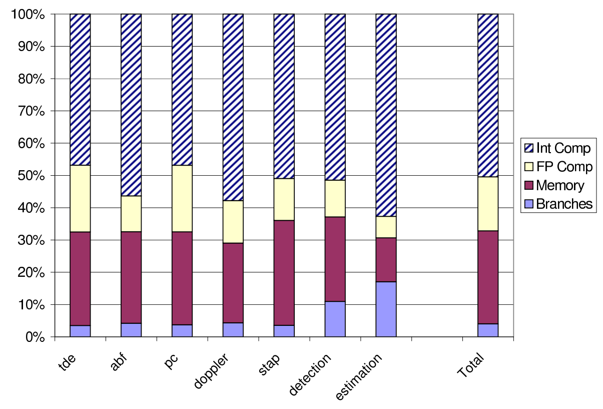

Figure 6: Instruction mix of the seven stages of the application.

3.4 Computation requirements

Figure 7 shows the average number of floating point operations performed in each of the stages. We can see that almost 66% of them are performed by the filtering stages (*tde* and *pc*).

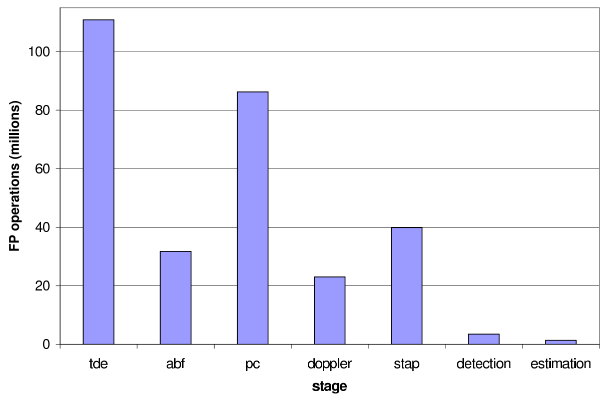

Figure 7: Average number of floating point operations required by each stage.

Given the real-time nature of the GMTI application, the entire data cube needs to be processed by the seven stages in a certain amount of time. This time is known as the coherent processing interval. The stages perform a total of 295 million floating point operations per interval. The coherent processing interval depends on the configuration parameters of the radar system. Our experiments use an interval of 8.5 ms, hence requiring 35 GFLOPS.

3.5 Memory behavior

Figure 8 classifies the memory accesses performed by the GMTI stages in the Grid system. It shows that most of the accesses are handled by the L1 data cache, except for the *abf* and *stap* stages. This behavior is explained by the access patterns described in Figure 3. We observe that stages *tde* and *pc* access a single vector of size *range*. In our configuration this vector has 2691 elements, and occupies 21 KB in memory, so it is small enough to fit in the L1 data cache. Stages *abf* and *stap* iterate through the data cube in 2 dimensions at a time, they access more data, and hence depend more on the L2 cache. The data cube used in the *stap* stage is larger, so it results in a substantial amount of memory accesses.

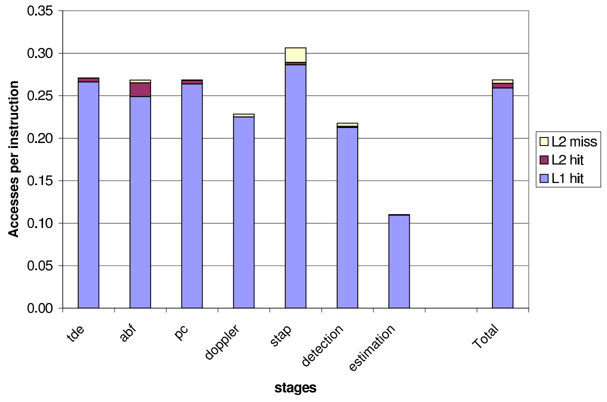

Figure 8: Cache hits/misses per instruction.

3.6 Parallelism

The results shown until now focus on the base Grid configuration. Here, we explore the way in which the stages make use of the available parallelism in the machine. Figure 9 shows the performance of the different GMTI stages as we increase the machine level parallelism in the base superscalar processor. The computation resources – including number of functional units, number of reorder buffer entries and load/store queue entries – have been scaled accordingly.

As mentioned earlier, stages *tde* and *pc* benefit more from the available parallelism. But even for those stages, the benefit of an increased width in the superscalar configurations decreases after 8. This is due to the difficulty in finding independent instructions with a reduced instruction window.

The Grid processor tries to prevent this problem. For example, the base grid processor used in our experiments has 128 physical frames, which allow it to have 2048 on-flight instructions, compared to the 192 possible in the superscalar case (128 entries in the reorder buffer and 64 in the reservation stations). Figure 10 shows the performance for different Grid systems in terms of their instructions-per-cycle (IPC). In

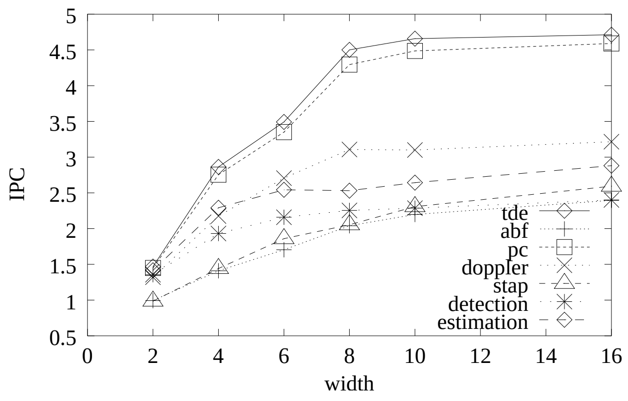

Figure 9: IPC for each of the GMTI's stages in a uniprocessor system.

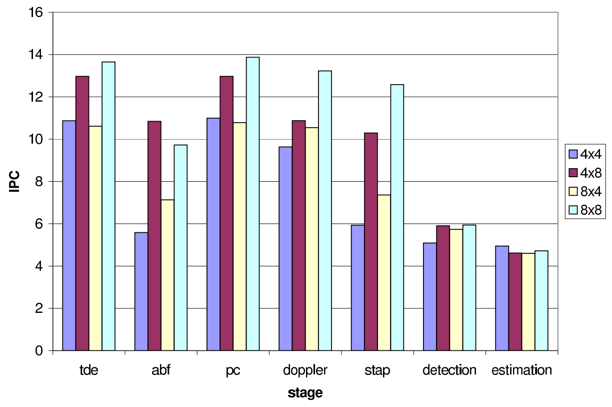

Figure 10: IPC for each of the GMTI's stages in a TRIPS system. Systems are labeled as $width \times height$, based on the diagram shown in Figure 4. The sizes of the register file and caches are maintained constant.

general, the application benefits from the parallelism offered by the hardware. The exception is the 8x4 configuration, which has the same number of functional units as the 4x8 configuration, but a significantly lower IPC. These results are due to the longer path from the functional units to the data cache (see Figure 4).

Another problem observed in our experiments is that the *abf* stage shows a better IPC with the 4x8 configuration than with the 8x8 configuration. This problem is due to the scheduler and has been reported by Sankaralingam et al. [12]. But we observe that the grid system scales well for this application. The early stages of the GMTI application see a noticeable performance improvement when going from the 4x4 grid configuration to the 8x8 grid. They are able to exploit the parallelism of the system more effectively, and we see speedups of up to $2.2x$ (in the *stap* stage). The last two stages (*detection* and *estimation*) have less data parallelism and depend more on control flow and, so they do not appear to benefit from the increased number of functional units.

4 Conclusions

Computer architects and designers are constantly trying to understand the behavior of emerging and future computer workloads. This paper studies the execution of the ground radar motion indication (GMTI) workload on two different types of processor architectures. The workload contains several processing stages very rich in matrix multiplications and fast Fourier transforms, very typical of workloads in the physical, chemical and biological sciences. The GMTI application has huge computation demands and can be considered to be typical of future high performance computing applications.

We studied the execution of this application on an ILP (superscalar) processor and an emerging grid processor architecture. The application is seen to contain a large amount of parallelism that can be exploited by spatially or temporally parallel computer architectures. However, the non-uniformities in computing requirements, as well as memory access patterns of the different stages, are important considerations in the design of spatially/temporally parallel architectures to handle these applications. The regularity of this class of applications is encouraging while parallel architectures are designed for these workloads, but the variations in requirements of different stages become a challenge in pipelining the stages.

Acknowledgments

We want to thank Matt French for his help on setting up the application and the anonymous reviewers for their suggestions to help improve the quality of this paper. This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-1892, NSF grant ECS-0113105 and by Tivoli, Intel and IBM Corporations.

References

[1] S. Keckler, B. O'Krafka, A. Kumar, R. S. Pearlman, and R. Rajamony, "Panel: Future workload predictions and implications for computer system design," in *Proceedings of the 6th International Workshop on Workload Characterization (WWC-6)*, (Austin, TX, USA), IEEE, Oct. 27 2003.

[2] "SPEC CPU2000 benchmark." **http://www.specbench.org/osg/cpu2000**.

[3] J. D. McCalpin, “STREAM benchmark.” **http://www.cs.virginia.edu/stream/**.

[4] “The polymorphous computing architecture (PCA) initiative.” **http://www.darpa.mil/ipto/programs/pca/**.

[5] J. Lebak, J. McMahon, and M. Arakawa, “Polymorphous computing architecture (PCA) application benchmark 1: Three-dimensional radar data processing,” tech. rep., MIT Lincoln Laboratory, Mar. 1 2004.

[6] J. M. Lebak, “Preliminary design review: PCA integrated radar-tracker application,” Tech. Rep. PCA-IRT-1, MIT Lincoln Laboratory, Apr. 9 2002.

[7] J. Rahe and S. W. Keckler, “Analysis of polymorphous computing architecture (PCA) radar-processing benchmark,” Tech. Rep. TR-03-41, University of Texas at Austin, Department of Computer Sciences, Oct. 2003.

[8] H. R. Raemer, *Radar Systems Principles*. Boca Raton, FL, USA: CRC Press, 1996.

[9] A. Reuther, “Preliminary design review: GMTI narrowband processing for the basic PCA integrated radar-tracker application,” Tech. Rep. PCA-IRT-2n, MIT Lincoln Laboratory, Feb. 24 2003.

[10] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*. Upper Saddle River, NJ, USA: Prentice-Hall, 2 ed., 1998.

[11] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, “A design space evaluation of grid processor architectures,” in *Proceedings of the 34nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, (Austin, TX, USA), Dec. 3–5 2001.

[12] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore, “Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA 2003)*, (San Diego, CA, USA), June 9–11 2003.

[13] V. Kathail, M. Schlansker, and B. R. Rau, “HPL-PD architecture specification: Version 1.1,” Tech. Rep. HPL-93-80(R.1), Hewlett-Packard Laboratories, Feb. 2000.

[14] E. Larson, S. Chatterjee, and T. Austin, “MASE: A novel infrastructure for detailed microarchitectural modeling,” in *Proceedings of 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, 2001.

[15] D. Burger and T. M. Austin, “The simplescalar tool set, version 2.0,” Tech. Rep. TR-1342, Dept. of Computer Science, Univ. of Wisconsin-Madison, June 1997.

[16] S. W. Keckler, D. Burger, C. R. Moore, R. Nagarajan, K. Sankaralingam, V. Agarwal, M. Hrishikesh, N. Ranganathan, and P. Shivakumar, “A wire-delay scalable microprocessor architecture for high performance systems,” in *International Solid-State Circuits Conference (ISSCC)*, pp. 1068–1069, 2003.

[17] D. Talla, L. John, and D. Burger, “Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements,” *IEEE Transactions on Computers*, vol. 52, pp. 1015–1031, Aug. 2003.