Flexible Filters for High-Performance Embedded Computing

Rebecca L. Collins and Luca P. Carloni
Department of Computer Science
Columbia University
{rlc2119,luca}@cs.columbia.edu

1. Introduction

Many high-performance embedded applications can be naturally programmed with a stream-processing model which exposes the inherent locality and concurrency among their tasks and enables efficient implementations on parallel architectures. Since an embedded software application may be characterized by nondeterministic and data-dependent behavior (as both the execution times of its tasks and their data token production/consumption rates may vary dynamically) efficient implementations on multi-core processor systems require load-balancing techniques. We propose *flexible filters* as a technique to dynamically balance the load in stream programs when bottlenecks in the flow of data arise [4]. Flexible filters can be added to a program without modifying any of the original user code, and introduce very little runtime overhead, depending only on backpressure signals that are already present in the buffer APIs.

2. Flexible Filters

A stream program is a parallel program broken up into a pipeline of tasks, called *filters*, connected with communication channels. A *bottleneck filter* is a filter that becomes a bottleneck for data tokens in the stream, e.g. because its operation has a significantly higher computational cost than the other filters. A bottleneck filter may be transient, resulting from data-dependent special-case handling, or constant, due to an inherently more expensive task. In either case, when a filter is a bottleneck, optimal system utilization is lost because cores executing upstream filters must stall on full communication buffers, and cores executing downstream filters are starved for data. Fig. 1 shows a simple stream program consisting of three filters connected in a pipeline, with each filter mapped to a separate processing core. Suppose that filter C becomes a bottleneck and cannot keep up with the data tokens being sent through the stream by A and B. As a result, core$_3$ must force core$_2$ to stop. If this imbalance continues, core$_2$ will eventually need to send a stop signal to core$_1$ as well.

Flexible filters alleviate bottlenecks at runtime by recruiting idle upstream cores to help with the work of the bottleneck filter. Suppose that core$_2$ can also execute filter C, as illustrated in Fig. 2. Then, instead of stalling, core$_2$ can "work ahead" on the data tokens waiting in its buffers. Now the rate at which data tokens are processed by filter C is increased, and core$_3$ has fewer data tokens to process, and so the system can run faster. Filter C is duplicated on core$_2$ so that core$_2$ can share core$_3$'s load. Two auxiliary filters, *flex_split* and *flex_merge* (represented as small black boxes in Fig. 2), orchestrate the split and merge of the data stream based on available space in the downstream buffers. The code overhead of *flex_split* and *flex_merge* is very low since backpressure signals from the buffers are already in use in FIFO based pipelines to prevent buffer overflow. They can be added to the stream program without changing any of the original filters, and guarantee preservation of the order among the data tokens in the stream.

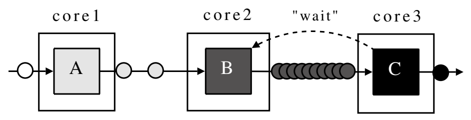

Figure 1: Bottleneck Filter in a Pipeline Mapping.

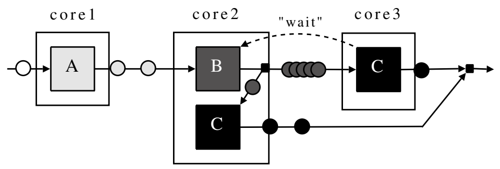

Figure 2: Flexible-Filter Mapping.

Many stream programming languages, such as StreamIt, include *split* and *join* nodes in their set of library functions [5]. The compiler may also insert *split* and *join* in order to optimize the program by increasing data parallelism. This achieves static load balancing because the data flow is split at run-time regardless of the loads on the various cores. The flexible-filters approach starts with a static mapping of filters to cores optimized based on profiling of the application but achieves dynamic load balancing that adapts to load imbalances which may be temporary or variable. The combination of static and dynamic techniques is also used by Flextream, which performs static compilation based on a virtualized multi-core system, and runtime (re-)assignment of filters to provide portability across platforms while tolerating changes in available system resources [7]. Chen *et al.* propose an alternative, where several alternative filter mappings are compiled and the runtime system "context-switch"es between them [3]. Context-switches in the stream mapping are complementary to flexible filters, applicable to larger scale changes in flow behavior.

Stream-specific load balancing techniques allow for the optimization of buffer space allocation and minimize unnecessary code and data movement. Work stealing is a popular general purpose load balancing technique that balances load by allowing idle cores to "steal" tasks from busy cores [2, 9]. Most work stealing techniques go through stages of load evaluation, reassignment, and task migration; and their "victim" processors (from whom tasks will be stolen) are selected randomly. In contrast, flexible filters do not steal randomly, but use the knowledge that neighbors of a bottleneck filter will be idle because they dependent on this filter to continue processing data tokens. Moreover, the decision to redirect flow is always made locally.

3. Experiments

We implement flexible filters using the Gedae dataflow language and test a set of benchmarks on the Cell BE processor [1, 8], including the constant false-alarm rate (CFAR)

Benchmark	Input Data	Speedup
CFAR	% targets/workload	
	7.3/16μs	1.45
	7.3/32μs	1.39
	7.3/63μs	1.47
	1.3/16μs	0.82
	1.3/32μs	1.06
	1.3/63μs	1.27
Dedup	Rabin block/max chunk size	
	4096/512	2.00
JPEG	image width x height	
	128x128*	1.31
	256x256*	1.16
	512x512*	1.25
Value-at-Risk	stocks/walks/timesteps	
	16/1024/1024	0.98
	64/1024/1024	1.56
	128/1024/1024	1.55

*individual benchmark images, each with distinct content

Table 1: Summary of speedup results for benchmarks where one bottleneck filter is made flexible.

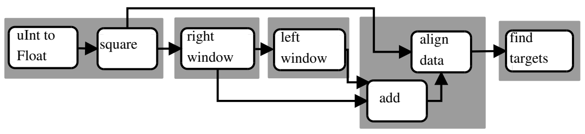

Figure 3: CFAR block diagram.

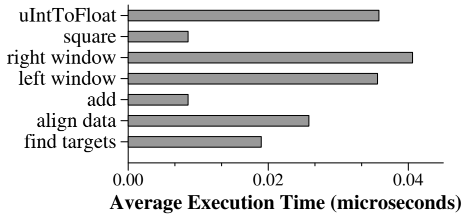

Figure 4: Profile of CFAR filters on Cell.

detection benchmark from the HPEC Challenge [6]. Table 1 lists the speedup gained in several stream benchmarks by flexible filters compared to a parallel stream implementation without flexible filters. Next we analyze the CFAR benchmark in more detail. CFAR detection identifies targets in a stream of incoming data with a noisy background, using an adjustable threshold that changes based on the background noise so that the false alarm rate is constant. A block diagram of the CFAR benchmark is shown in Fig. 3, and Fig. 4 shows a profile of CFAR's filters' execution times in our implementation. All of CFAR's filters have a relatively small execution time with respect to the communication overhead, and we initially found no benefit to adding flexibility to the program. In particular, the *find targets* filter in our original implementation does not do additional work after a target is detected, and so has relatively constant execution time regardless of the content of the data stream. However, in practice it is possible that once a target is found, additional processing such as target classification and tracking is desired [10]. To capture this fact, in the CFAR experiments reported in Table 1 additional work is performed every time a target is detected. Since the location of targets is data dependent and may not be uniformly distributed in the stream, the workload of *find targets* may change dynamically, and spikes in the number of targets detected could cause bottlenecks. Fig. 5 plots a histogram of the time it takes to process a data block of 114 cells, where 7% of the cells are targets, and an additional workload of 32μs is added for each target. The speedup gained by applying flexible filters in this case depends both on the percentage of data tokens that require extra processing and on the amount of extra work required. Notice that flexible filters may introduce some runtime overhead because they need a few additional buffers with respect to a non-flexible filter mapping. This reduces the amount of memory for the original buffers and may force a finer granularity of the data blocks to be transferred, which can affect the overall application performance.

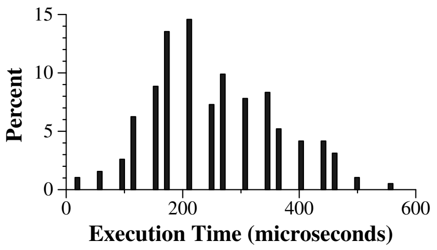

Figure 5: Histogram of workload per 114 cells, with % targets/workload = 7/32μs.

4. Conclusions

As the scale of multi-core systems increases, the number of concurrent tasks within a single application will also grow as programmers attempt to extract as much available parallelism as possible. With increasing parallelism at the application level, unbalanced tasks and dynamic load variations become more likely. Flexible filters provide a lightweight and effective mechanism to better leverage multi-core architectures for the execution of stream programs by making them adapt to the presence of data-dependent behavior.

References

[1] Gedae, http://www.gedae.com/.

[2] M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems*, 35(3):289–304, 2002.

[3] J. Chen et al. A reconfigurable architecture for load-balanced rendering. In *Proc. of the SIGGRAPH/ EUROGRAPHICS Conf. on Graphics Hardware*, pages 71–80, July 2005.

[4] R. L. Collins and L. P. Carloni. Flexible filters: Load balancing through backpressure for stream programs. In *Intl. Conf. on Embedded Software (EMSOFT)*, Oct. 2009.

[5] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of the Intl. Conf. on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, Oct. 2006.

[6] R. Haney, T. Meuse, J. Kepner, and J. Lebak. The HPEC challenge benchmark suite. In *The Ninth Anuual High-Performance Embedded Computing Workshop (HPEC 2005)*, Sept. 2005.

[7] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 214–223, 2009.

[8] J. Kahle et al. Introduction to the CELL multiprocessor. *IBM J. Res. Develop.*, 49(4-5):589–604, Sept. 2005.

[9] P. Kakulavarapu, O. Maquelin, J. N. Amaral, and G. R. Gao. Dynamic load balancers for a multithreaded multiprocessor system. *Parallel Processing Letters*, 11(1):169–184, 2001.

[10] L. M. Novak, G. J. Owirka, W. S. Brower, and A. L. Weaver. The automatic target-recognition system in SAIP. *The Lincoln Laboratory Journal*, 10(2):187–202, 1997.