Intra-Application Data-Communication Characterization

Imran Ashraf, Vlad-Mihai Sima, and Koen Bertels

Computer Engineering Lab, Delft University of Technology, The Netherlands
{I.Ashraf,V.M.Sima,K.L.M.Bertels}@TUDelft.nl

Abstract. The growing demand of processing power is being satisfied mainly by an increase in the number of computing cores in a system. One of the main challenges to be addressed is efficient utilization of these architectures. This demands data-communication aware mapping of applications on these architectures. Appropriate tools are required to provide the detailed intra-application data-communication information and highlight memory-access patterns to port existing sequential applications efficiently to these architectures or to optimize existing parallel applications. In this work, we present an efficient data-communication profiler which provides a detailed data-communication profile for single and multi-threaded applications. In contrast to prior work, our tool reports such information with manageable overheads for realistic workloads. Experimental results show that on the average, the proposed profiler has at least an order of magnitude less overhead as compared to other state-of-the-art data-communication profilers for a wide range of benchmarks.

Keywords: Memory-profiling, Data-communication characterization, Heterogeneous computing, communication-aware mapping, shadow memory

1 Introduction

Although transistor scaling yields to more transistors per chip, inherent physical limits prevent further cost-effective down scaling due to multiple challenges such as increased power consumption and complex fabrication process [13]. As a result, designers have shifted the computational paradigm by integrating more and more homogeneous and heterogeneous processing cores in general-purpose (Kaveri by AMD), embedded (Zynq by Xilinx) and high-performance computing platforms (Hybrid-core by Micron). This emerging trend poses specific challenges regarding their programmability as an effective use of these platforms demands architecture-aware application mapping.

To exploit core-level parallelism, applications must be divided into smaller parts which are mapped to the available cores in the architecture. This is a critical task as an improper partitioning may diminish the anticipated performance improvements. The main identifiable reasons for such performance degradation are the inefficient memory assignments and huge inter-core data-communication overhead [9]. With the increasing number of cores the degradation in performance exacerbates as this communication is typically more time-consuming than computation. Hence, it is considered as the major design challenge in multi-core architectures [16]. In addition, it is a major source of energy consumption [4].

Efficient application partitioning demands the understanding of the data-flow in an application. With the growing program complexity, driven by an increasing demand of processing, it is time-consuming, tedious and error-prone to manually analyze these complex applications. Hence, program analysis tools are required to identify the hot-spots and/or bottlenecks pertaining to the target platform [16].

Well maintained open-source and commercial performance analysis tools exist which report communication in programs where communication is explicit, such as MPI [5, 6]. We would like to highlight here that these tools are not designed to provide the communication profile of sequential applications. These tools are based on the technique which requires MPI parallel program as input. Hence, it only helps in validating the parallel program written only in MPI, rather than constructing one.

Data-communication profilers based on static-analysis tools can be developed [10], but they can only be used for regularly-structured applications and are inapplicable for most of the real-world programs due to their irregular structure. Additionally, pointer-analysis and their dynamic nature makes it hard to track the data-communication statically. Hence, dynamic methods are required to characterize the data-communication for such programs. Dynamic analysis tools generally have a high overhead as compared to static ones. To generate a realistic profile of applications dynamically requires the use of realistic workloads, which results in an increase in overhead. Another challenge with such tools is the difficulty of pinpointing the exact source-code location that is responsible for the data-communication. This information, though very useful for developers, makes the design of such tools challenging and increases their overhead.

In this work, we present an open-source memory-access and data-communication profiler which addresses these issues. The proposed tool provides detailed information which is not provided by existing tools. As a case-study, we analyze memory-access patterns and data-communication bottlenecks for a feature tracking application. Experimental results show that the proposed tool generates this information at considerably reduced execution-time and memory-usage overhead due to its well-thought design. The main contributions of this work can be summarized as follows.

- The design of a novel data-communication profiler.
- The analysis of memory-accesses and data-communication behavior of a feature tracking application as a case-study. Subsequently, this information is utilized to port this application to a GPU based platform.
- A comparison with the state-of-the-art focusing on two criteria: execution-time and memory-usage overhead.

The remainder of this paper is structured as follows. We start by providing the related work in Section 2. The design of the proposed profiler is detailed in Section 3. In Section 4, we discuss the use of generated information as a case-study to map an application on a platform using GPU as accelerator. An empirical comparison of the overheads is presented in Section 5, followed by conclusions in Section 6.

2 Related Work

Various open-source [7, 17, 20] and proprietry [2, 3] tools exist which perform memory profiling. However, these tools only provide the information about the cache misses and do not report data-communication information in an application to perform partitioning or communication-aware mapping.

Though static-analysis tools [10] can also track data-communication, a large number of tools utilize dynamic-analysis to collect accurate information at runtime. A well-known dynamic-analysis technique used by large number of tools is instrumentation. Various tools based on this technique are used for finding memory-management [22] and threading bugs [21,22]. However, very few tools exist, as discussed in this section, which perform detailed data-communication characterization, especially in sequential applications for efficient application partitioning.

Redux [18] a Valgrind based tool, draws the detailed Dynamic Data-Flow Graphs (DDFGs) of programs at the instruction-level. This tool has huge overhead as it generates fine-grained DDFGs. Hence, it can only be used for very small programs or parts of programs, as discussed by authors. Secondly, the purpose of the tool as reported by authors is to represent the computational history of a program and not to report its communication behavior.

Pincomm [12] reports the data-communication and it it is based on Intel Pin Dynamic Binary Instrumentation (DBI) framework [15]. Pincomm uses a hash-map to record the producer of a memory location. Due to this map, the tool has high memory overhead. Due to this overhead, Pincomm stores the intermediate information to the disk and reads it later by a script to generate the communication graph. This disk writing incurs high execution-time overhead. Furthermore, the authors also mention the use of markers in the source-code to reduce the overhead and manage the output complexity. However, in complex applications inserting these markers manually is time-consuming. Secondly, this marking requires knowledge of the application to understand what are the important parts of the program, which is not trivial.

Quad [19], also based on Pin [15], provides data-communication information between functions by tracking memory access at byte-granularity. Trie is used to store producer-consumer relationships and it does so with less memory overhead as memory allocations in the Trie are done on demand at granularity of each byte. However, this approach has high execution-time overhead, mainly because of the access-time of Trie and the frequent memory allocations. Furthermore, the cumulative information is reported at the application-level, which makes it difficult to utilize. In addition, the information generated is not really useful when the application has different memory access behavior per call. Moreover, the provided information is without suitable relationship to the application source-code, which makes its use tedious for developers.

Summarizing, existing approaches have high execution-time and memory-usage overhead, which limits their use for realistic workloads. This may affect the quality of the generated profile. Furthermore, the provided information lacks necessary dynamic details and is not linked to the source-code, making it hard to utilize this information.

3 MCPROF: Memory and Communication PROFiler

In this section, we present the design of mcprof [1] which can conceptually be divided into three main blocks as depicted in Figure 1 and detailed in this section.

3.1 Memory Access Tracer

The memory access tracer uses Intel's Pin [15] DBI framework to trace memory reads and writes performed by the application. We utilize instruction-level instrumentation

[1] `https://bitbucket.org/imranashraf/mcprof/downloads`

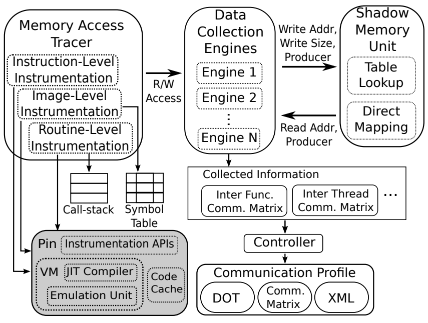

Fig. 1: Block Diagram of MCPROF.

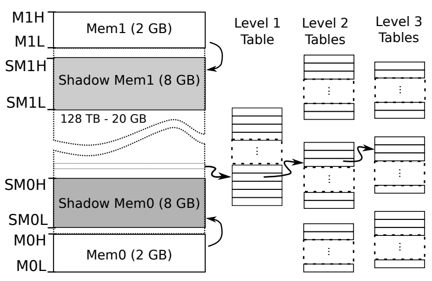

Fig. 2: Hybrid Shadow Memory Scheme Utilized by MCPROF.

to track memory reads and writes by each instruction. Furthermore, routine-level instrumentation is utilized to keep track of the currently executing function. These are tracked by maintaining a call-stack of the functions executing in the application. Static symbols are obtained by reading Executable and Linkable Format (ELF) [8] header. To track the dynamic allocations, image-level instrumentation is utilized to selectively instrument library images for memory (re)allocation/free routines.

An important point to mention here is that although in the current implementation we have used the Pin framework to trace memory accesses, in the future, if desired, with minor modifications, it is possible to use any other DBI framework or any other technique to trace memory accesses.

3.2 Data Collection Engines

On each memory access traced by the Memory Access Tracer, a specific callback function is triggered based on the selected engine. In the case of a write, the producer of the memory address is recorded in the shadow memory. On a read access, the producer is retrieved from the shadow memory, while the consumer of the memory access is the function at the top of the call-stack. Furthermore, based on the information required by each engine, extra information is also recorded. For instance the source-line and filenames of the allocated blocks as well as the allocation size, which is stored in the symbol-table. Currently we have implemented the following three engines in MCPROF.

Engine-1: This engine reports the memory-intensive functions and objects in the application. This information, combined with the execution profile of the application, can be automatically used, if desired, to reduce the overhead by performing selective instrumentation and also reduce the complexity of generated profile.

Engine-2: This engine records inter-function/inter-thread data-communication at the application level. The data-communication information is stored in a data-communication matrix, where indices of the matrix are the producer and consumer function/threads. When object-tracking is enabled, the data-communication is reported to/from the objects in the source-code.

Engine-3: This engine generates per-call data-communication information. This is important for applications with irregular memory access behavior per-call. Each call is also given a unique sequence number which helps in identifying the temporal information of each call.

3.3 Shadow Memory

This block is responsible for recording the producer of each byte. On each write access, the selected engine sends the address, size, thread ID and the function at the top of the stack, which is the writer (producer) of this byte to the shadow memory unit. When a function reads a byte, the reader (consumer) is the currently executing function, while the the producer is retrieved from the shadow memory unit. These reads and writes can happen anywhere in the 128 TB user address space, so keeping track of the producer efficiently, is not trivial. Hence, the design of this shadow memory block has a great impact on the execution-time and memory-usage overheads of a profiler. Hence, we have combined the following two techniques in the design of the shadow memory unit.
Direct Mapping in which an application's address is translated to a shadow memory address by using a *Scale* and *Offset*. Given an address *Addr*, its shadow address will be $(Addr \times Scale) + offset$. Although this address translation is fast, it assumes a particular OS memory layout and requires the reservation of a huge amount of virtual memory at fixed addresses.
Table-lookup in which multi-level tables are used to map addresses in an application to their shadow addresses. This is similar to the page look-up tables utilized in OSes. This approach is more flexible as it does not require neither a fixed memory layout, nor an initial reservation of huge memory, as tables are allocated on demand. The downside of this approach is that the multi-level table look-up is slower than the address translation in direct mapping.

In order to make a well-informed trade-off between flexibility, execution-time and memory-usage overheads, we have utilized a hybrid design of the shadow memory unit as shown in Figure 2. We analyzed the access frequency in the memory map and found out that the most frequently accessed memory is the bottom (Mem0) and the top (Mem1) regions in the memory map. For most of the applications, N and M can be 2 GB as shown in Figure 2. To make accesses to these regions faster, we reserve[2] in advance two shadow memories corresponding to these two memory regions, shown as *Shadow Mem0* and *Shadow Mem1*, respectively. This results in a simpler mapping of addresses in these regions to the shadow addresses by Equation (1), without requiring any lookup.

$$Addr_{sh} = ((Addr \& M0H) << log_2(SCALE)) + (Addr \& (SM1L + SM0L)) + SM0L \quad (1)$$

where, $Addr$ is the address of the original byte, $Addr_{sh}$ is the address of the corresponding shadow bytes, $SCALE$ is 4, and $M0H$, $SM1L$ and $SM0L$ are constants as shown in Figure 2. For the middle (128 TB − 20 GB) less frequently used region, we utilize a 3-level table-lookup scheme as shown in Figure 2.

Initially, the level-1 table is created and all its entries are marked as *UNACCESSED*. Tables in the remaining two levels are created on demand when the address in that range is touched for the first time. The address of the memory accessed in this region is used to index these tables to reach level-3 where 4 shadow bytes are written for each byte memory accessed in the original program. One byte for the function ID, one byte for thread ID and 2 bytes for the ID of the object this address belongs to. Therefore, we currently restrict the number of function and thread IDs to 256. In the

[2] These regions are only reserved in the memory map, actual memory-usage is 4 B for each byte of memory used by the program.

future we will investigate more applications, and if required, increase the number of bytes to store the IDs, as it is simply a parameter in the tool.

4 Case-study

The focus of this case study is on the utilization of data-communication information provided by MCPROF to map an application onto the GPU, without performing algorithmic modifications. The use case involves Kanade-Lucas-Tomasi Feature Tracker (KLT) application [14]. This application detects interesting features in a frame and tracks them in the subsequent frames. We have used version 1.3.4, which is the latest version of KLT [1]. This *C* implementation has 102 functions in 17 source-files making up 5033 lines of code.

For the experiments performed in this case study, we used 64 bit, 2.5 GHz Intel(R) Xeon(R) CPU with 32 GB RAM. Nvidia GeForce GT 640 GPU, with 2 GB memory, is used as an accelerator which is connected to the PCIe slot of the CPU. Ubuntu 12.04 is running on the machine with Linux kernel 2.6.32-24-server and Nvidia driver version 319.37. Nvidia CUDA toolkit $V6.0$ is used to program the GPU.

4.1 Implementation without Data-communication Optimization

In order to efficiently map an application onto an accelerator based platform, compute intensive functions, known as kernels, are off-loaded to the accelerator. We used *gprof* [11] to identify the kernels in the application as shown in Table 1. For this run, 30 frames have been used with frame size chosen as 1024×768, to accumulate enough number of samples to generate representative profile of the application. The total percentage contribution of these kernels 91.7%.

As a first step in the mapping process, we mapped these kernels to the GPU. Table 2 provides the timing results of the first mapping step. For these experiments, 1024 features were tracked from frames of size 1024×768. Column 1 contains the names of the compute-intensive kernel. Column 2 lists the execution-time of these kernels on CPU (t_{cpu}) in seconds. $t_{gpu_{comp}}$ is the time spent in performing the computation on GPU which is shown in Column 3. The communication time $t_{gpu_{comm}}$ is listed in Column 4 which is the time spent in transferring data to GPU before computation and reading the results back, after the computation is complete. The execution-time speedup is the ratio of the execution-time on CPU and GPU. Total kernel speedup ($S_{K_{total}}$) is reported in Column 5, which is calculated as $\frac{t_{cpu}}{t_{gpu_{comp}}+t_{gpu_{comm}}}$. In order to highlight the effect of data-communication, Column 6 lists the kernel speedup ($S_{K_{comp}}$) for only the computation, calculated as $\frac{t_{cpu}}{t_{gpu_{comp}}}$.

From Column 5 in Table 2, it can be seen that speedup has been obtained for all the kernels except for `trackFeature` kernel. Hence, this kernel should not be mapped to GPU. Another important result that can be deduced by comparing Column 5 to Column 6 is that the communication has significantly reduced the achieved speedup. In the next sub-section we will perform the optimization of this data-communication by utilizing MCPROF.

Table 1: *gprof* flat profile for the *KLT* application.

Function Name	%Time
KLTSelectGoodFeatures	54.07
convolveImageVert	19.65
convolveImageHoriz	10.17
trackfeature	7.81
%Total Contribution	**91.7**

Table 2: Execution Time (sec) and Speedup results for the initial KLT implementation.

Kernel	t_{cpu}	$t_{gpu_{comp}}$	$t_{gpu_{comm}}$	$S_{K_{total}}$	$S_{K_{comp}}$
KLTSelectGoodFeatures	13.53	1.17	0.36	8.8×	11.52×
convolveImageVert	3.93	0.14	0.76	4.35×	28.08×
convolveImageHoriz	1.77	0.18	0.76	1.87×	9.89×
trackFeature	1.96	1.49	0.52	0.96×	1.31×

Table 3: Memory Intensive Objects in *KLT* reported by MCPROF.

Objects	Reads	Writes	Reads/Writes	Total	%Total
tmpimgCS	3.8e8	5.1e7	7.4	4.3e8	26.6
pointlist	1.3e8	1.3e8	1	2.6e8	16.3
pyramidImg	1.3e8	3.5e7	3.8	1.7e8	10.3
grady	1.34e8	3.1e6	42.7	1.3e8	8.3
gradx	1.34e8	3.1e6	42.7	1.3e8	8.3
tmpimgTF	6.7e7	9.4e6	7.1	7.6e7	4.6
guassderiv_kernel	6.7e7	4.7e3	14063.1	6.7e7	4.1
guass_kernel	6.7e7	4.6e3	14500.6	6.7e7	4.1
%Total Contribution					**82.6**

4.2 Optimization of Data-communication

Optimization of the data-communication requires understanding of the data-flow in the application. Understanding this data-communication by manual source-code is not trivial, especially for applications like the one we have considered in this case study involving a large number of functions and objects. Furthermore, pointer arithmetic exacerbates this problem making it is hard to determine the real producer and consumer of the data.

MCPROF provides this production-consumption information in the form of a data-communication graph in various formats. An overview of this information is shown as communication matrix in top right corner of Figure 3 representing inter-function communication intensity. MCPROF also generates the detailed quantitative data-communication information in the form of a directed graph. As there are large number of functions in the KLT application, the complete graph is too large to present here. Secondly, such large graphs are hard to be utilized by developers. Typically, the most compute-intensive functions are selected for analysis. MCPROF detects memory-intensive objects and the functions communicating with these memory-intensive objects. Table 3 lists the memory-intensive objects of the KLT application reported by MCPROF. Apart from mentioning the reads and writes accesses, the percentage accesses are also reported in the last column. The last row of the table shows that memory accesses through these 9 objects correspond to 82.6% of the total application memory accesses.

Figure 3 shows the data-communication graph of KLT application generated by MCPROF while tracking 256 features in 3 frames of size 1024 × 768. The ovals represent the functions in the application whereas the objects are represented by rectangles where the number inside the rectangle is the allocation size. The kernels in the application are shown in Grey ovals. The amount of communication in bytes is represented by directed edges, where the color of the edges represent the intensity of the communication. To simplify the discussion, the dotted lines are used to mark the functions in the main stages of applications.

`tmpimgSF` and `tmpimgTF` are generated by `KLTToFloatImage` on CPU and transferred to GPU as an input to `convolveImageHoriz`. `KLTToFloatImage`, though not

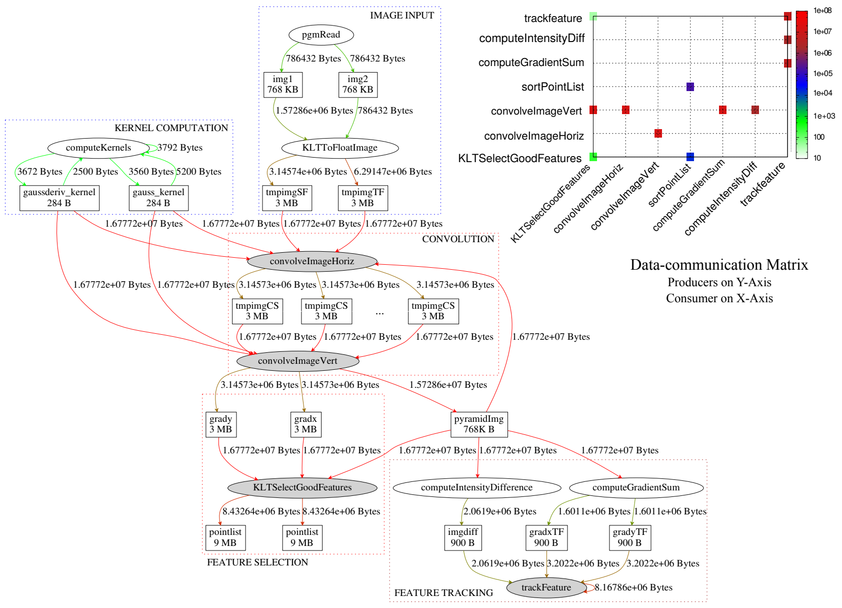

Fig. 3: KLT Communication Matrix (top right) and communication graph generated by MCPROF. Functions (ovals), compute-intensive functions (Grey ovals) and the objects(rectangles) involved in the communication are also shown.

compute-intensive, still mapping it to GPU will be better as it will make `tmpimgSF` and `tmpimgTF` internal to GPU and reduce CPU-GPU communication.

`guassderiv_kernel` and `guass_kernel` are generated by `computeKernels` on the CPU and consumed by `convolveImageHoriz` and `convolveImageVert` on GPU. However, mapping `computeKernels` to GPU is not required as `guassderiv_kernel` and `guass_kernel` are consumed heavily but produced very infrequently. This is also evident from the very high Reads/Write ratio (Table 3) implying very less production and a lot of consumption of data from these objects. Furthermore, these objects are very small in size (284 Bytes), hence can be easily mapped to GPU's constant memory.

Another optimization which can be performed in the convolution stage is the allocation and de-allocation of large number of `tmpimgCS` objects for each frame. Allocating a single object in the start and re-using it in the subsequent frames instead of re-allocating it will reduce the execution-time. Similar optimization can be performed for `pointlist` in the Feature Selection stage.

`gradx` and `grady` generated in the Convolution Stage are consumed in the Feature Selection stage by `KLTSelectGoodFeatures`. Hence, these objects can be kept on the GPU for utilization in these stages. Furthermore, these objects should be mapped to GPU's shared memory. This is because of high Reads/Writes ratio depicted in Table 3, suggesting high re-use of these objects. This will result in performance improvement as shared memory has higher bandwidth as compared to global memory. On the contrary, `pointlist` has Reads/Writes ratio of 1, which suggests no reuse, hence it should be kept in the global memory. Mapping `pointlist` to shared memory will only increase the overhead of the data transfer between global memory and shared memory without being reused.

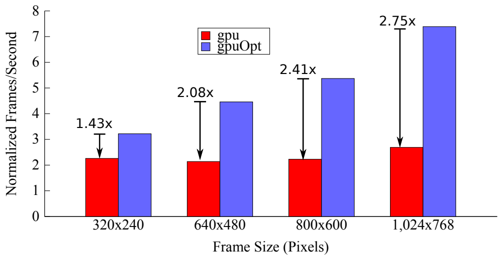

Fig. 4: Normalized Frames per seconds achieved by the GPU and data-communication optimized GPU implementation.

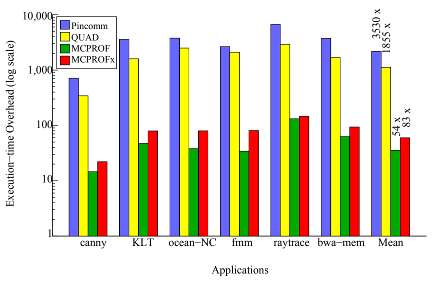

Fig. 5: Execution-time Overheads.

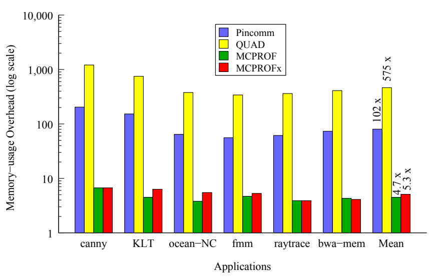

Fig. 6: Memory-usage Overheads.

Based on the preliminary results in Table 2, it was concluded that `trackFeature` should not be mapped to GPU because of slow-down. Even if this kernel is not so efficient on the GPU, we should still port it to the GPU to avoid the bulk of data-communication regarding the transfer of `pyramidImg` between GPU and CPU. This is clearly shown by the communication edges to the `computeIntensityDifference` and `computeGradientSum` function in the Feature Tracking stage.

After applying these optimizations to the initial GPU implementation (*gpu*), we obtained a data-communication optimized version of the GPU implementation (gpu_{opt}). Figure 4 shows the normalized Frames Per Seconds (fps) achieved by both the implementations for various frame sizes ranging from 320×240 to 1024×768 while the number of tracked features is set to 1024. Increasing the frame size, results in an increase in the amount of computation performed on the GPU. Increased computation results in better utilization of the available resources of the GPU, resulting in higher speedup as can be observed from this figure. On the other hand, increasing the frame size also increases the amount of frame data transferred to the GPU for processing and getting results back. This data-communication has been optimized in the case of gpu_{opt} based on the information provided by MCPROF. Hence, gpu_{opt} implementation achieves up-to 2.75× higher speedup as compared to *gpu* implementation where this communication is not optimized.

5 Overhead Comparison with Existing Profilers

In this section, we present the overhead comparison of MCPROF with other state-of-the-art data-communication profilers QUAD and PINCOMM as these are particularly designed to report data-communication. For these experiments, we used Pin

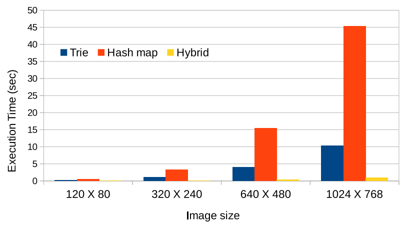

Fig. 7: Execution-time Comparison of Data-structure access only.

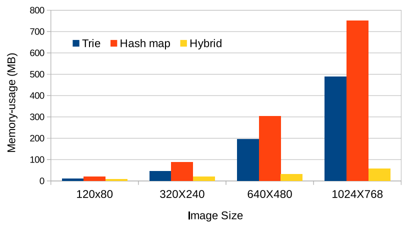

Fig. 8: Memory-usage Comparison of Data-structure access only.

v2.13 on the machine used in case-study. Figure 5 depicts the execution-time and memory-usage overhead of Pincomm, Quad and mcprof for applications from various domains, namely; image-processing (canny, klt) domain, SPLASH-2 benchmarks (ocean-NC, fmm, raytrace) and a bio-informatics application (bwa-mem). Each bar represents the ratios of the application execution-time with and without profiling for each profiler. Similarly, Figure 6 reports the ratios of application memory-usage with and without profiling.

We have reported mcprof results with two different settings depicted as *MCPROF* and *MCPROFx* in these figures. Results with *MCPROF* legend are overheads while providing the common basic information which Pincomm and Quad can also generate. Whereas, *MCPROFx* report overheads of complex engine while generating the detailed data-communication information with stack recording and object detection. Mean overhead results are also depicted in these figures. These results show that mcprof has, on the average, an order of magnitude less execution-time and memory-usage overheads. Main reason for this reduction in overhead is the well-thought design of the shadow memory scheme utilized by mcprof. Due to the design, we were able to shift most of the processing from analysis-time to instrumentation-time. Moreover, the access-time and memory-usage overhead of the hybrid shadow memory scheme is consisderably less as compared to Trie or Hash map utilized by Quad and Pincomm, respectively. In order to clearly illustrate this, we have plotted the execution-time and memory-usage of accessing only the data-structures of the three tools in Figure 7 and Figure 8, for the canny application for various image sizes.

Another important observation from Figure 5 is that mcprof has an average memory-usage overhead of $4.7 - 5.3\times$, which is mainly because 4 shadow bytes are allocated for each byte used in the original program, plus some additional memory for storing extra information.

6 Conclusions

Both the memory wall and the multi-core trend create the need for detailed data-communication profiling. In this work, we presented the design of mcprof, a memory-access and data-communication profiler. The unique design of the proposed profiler resulted in significantly reduced execution-time and memory-usage overheads as compared to the state-of-the-art, making the profiler useful for real applications with realistic workloads. The reduced overheads allowed us to generate additional memory-access and data-communication information which is not provided by existing tools.

References

1. KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker, http://www.ces.clemson.edu/~stb/klt/installation.html
2. Purify by IBM. http://www-03.ibm.com/software/products/us/en/rational-purify-family
3. vTune by Intel. http://software.intel.com/en-us/intel-vtune
4. Borkar, S., Chien, A.A.: The future of microprocessors. Commun. ACM 54(5), 67–77 (May 2011), http://doi.acm.org/10.1145/1941487.1941507
5. Brunst, H., Mohr, B.: Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with vampir NG. In: Mueller, M., Chapman, B., de Supinski, B., Malony, A., Voss, M. (eds.) OpenMP Shared Memory Parallel Programming, Lecture Notes in Computer Science, vol. 4315, pp. 5–14. Springer Berlin / Heidelberg (2008), http://www.springerlink.com/content/h704195763x61251/abstract/
6. Chung, I.H., Walkup, R., Wen, H.F., Yu, H.: Mpi performance analysis tools on blue gene/l. In: SC 2006 Conference, Proceedings of the ACM/IEEE. pp. 16–16 (Nov 2006)
7. Cohen, W.: Multiple Architecture Characterization of the Build Process with OProfile (2003), http://oprofile.sourceforge.net
8. Committee, T.: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 (May 1995)
9. Duato, J.: Beyond the power and memory walls: The role of hypertransport in future system architectures. In: WHTRA (February 2009)
10. Ernst, M.D.: Static and dynamic analysis: Synergy and duality. In: WODA 2003: Workshop on Dynamic Analysis. pp. 24–27. Portland, Oregon (May 9, 2003)
11. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A Call Graph Execution Profiler. SIGPLAN Not. 17(6), 120–126 (1982)
12. Heirman, W., et al.: PinComm: characterizing intra-application communication for the many-core era. In: ICPADS. pp. 500–507 (Dec 2010)
13. Horowitz, M., Dally, W.: How scaling will change processor architecture. In: ISSCC. pp. 132–133 Vol.1 (2004)
14. Lucas, B.D., Kanade, T.: An Iterative Image Registration Technique with an Application to Stereo Vision. pp. 674–679 (1981)
15. Luk, C., et al.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: PLDI '05. pp. 190–200. ACM, New York, NY, USA (2005)
16. Martin, G.: Overview of the MPSoC design challenge. In: 43rd ACM/IEEE DAC. pp. 274–279 (2006)
17. Nethercote, N.: Dynamic Binary Analysis and Instrumentation. Ph.D. thesis, University of Cambridge, United Kingdom (November 2004)
18. Nethercote, N., Mycroft, A.: Redux: A dynamic dataflow tracer. Electronic Notes in Theoretical Computer Science (2), 149–170 (Oct 2003)
19. Ostadzadeh, S.: Quantitative Application Data Flow Characterization for Heterogeneous Multicore Architectures. Ph.D. thesis, TU Delft (Dec 2012)
20. Pesterev, A., Zeldovich, N., Morris, R.T.: Locating cache performance bottlenecks using data profiling. In: Proceedings of the 5th European Conference on Computer Systems. pp. 335–348. EuroSys '10, ACM, New York, NY, USA (2010)
21. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: Data race detection in practice. In: WBIA. pp. 62–71. ACM, New York, NY, USA (2009)
22. Seward, J., Nethercote, N.: Using valgrind to detect undefined value errors with bit-precision. In: USENIX ATC. ATEC '05, Berkeley, CA, USA (2005)