Computation vs. Memory Systems: Pinning Down Accelerator Bottlenecks

Martha A. Kim and Stephen A. Edwards

***Abstract*—The world needs special-purpose accelerators to meet future constraints on computation and power consumption. Choosing appropriate accelerator architectures is a key challenge. In this work, we present a pintool designed to help evaluate the potential benefit of accelerating a particular function. Our tool gathers cross-procedural data usage patterns, including implicit dependencies not captured by arguments and return values. We then use this data to characterize the limits of hardware procedural acceleration imposed by on-chip communication and storage systems. Through an understanding the bottlenecks in future accelerator-based systems we will focus future research on the most performance-critical regions of the design. Accelerator designers will also find our tool useful for selecting which regions of their application to accelerate.**

I. Introduction

As researchers continue to devise compelling new computationally intensive applications, hardware systems have reached an uncompromising power wall that prevents any increase in system power budgets. Increased computational demands coupled with a fixed power budget demand advances in the performance per watt of tomorrow's chips. Although special-purpose hardware is the most efficient, we use it only when power and performance targets cannot be met in software.

We intend the tool we describe in this paper to assist designers of special-purpose hardware accelerators in answering two key questions: whether a function is computation- or memory-bound and, accordingly, which functions make good candidates for acceleration. Amdahl's law motivates the first question: accelerating something that consumes only 1% of the total execution time is obviously not worth the time it would take.

The main purpose of our tool is to help answer these questions. For a particular function does arithmetic dominates the work that a hypothetical accelerator must do or will it be dominated by data transfers to and from the accelerator? Matrix multiplication is a familiar computationally intensive task, but others are just as compelling yet place much larger (relative) demands on the memory system. Examples of these include solving Boolean Satisfiability problems, rotating images, discrete-event simulation, and many others.

We have developed a pintool [3] that collects statistics about a program's dynamic memory access behavior. Specifically, during the execution of a function, it identifies the function that most recently wrote each byte of data read by the currently running function. From this information, we derive a count of the number of bytes that flowed both into and out of each function, which is an estimate of how memory-intensive a particular task is.

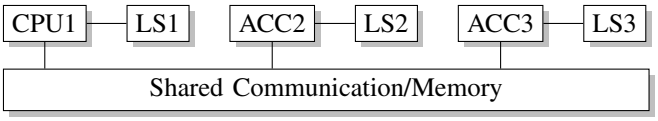


Fig. 1. Our model of accelerator hardware: A general-purpose CPU with its own local storage (e.g., a cache); accelerators, each with their own local storage; and a shared communication medium (e.g., a bus connected to memory) that the CPU and accelerators use to communicate. Our pintool strives to answer whether an accelerator or its interface to shared memory will be the bottleneck, determining where resources should be focused.

II. Our Vision for Accelerators

We built our tool to answer fundamental questions that will arise in designing and implementing our vision of custom hardware accelerators. Power is the motivation: to improve computational performance per watt, throwing general-purpose multicore processors at a problem is not the solution because they, necessarily, have much more hardware than strictly necessary to perform a certain computation.

From both power consumption and dissipation considerations, the chip of the future will have to consist of an enormous number of transistors; only a fraction of them can be powered on at any instant. We envision these chips will include a handful of general-purpose cores surrounded by many, many special-purpose accelerators that are powered on only when their function is needed.

Conceptually, an accelerator must do two things: perform computation and communicate with its environment. We envision future systems will continue to consist of processors communicating with memory through a shared bus of some sort (Figure 1); the communication we envision will thus be DMA-like, ultimately managed by software running on the general-purpose cores but conducted by the accelerators themselves.

To a programmer, we envision our accelerators will appear like libraries and thus present some sort of functional interface. Thus, accelerators will exhibit function-like behavior: each will need to gather input data from memory, operate on it, and finally write data back.

Our tool attempts to answer whether a particular accelerator is likely to be computation- or communication-bound, assuming this sort of functional interface. Our custom accelerator design flow starts with off-the-shelf unaccelerated software that performs the desired computation, such as JPEG decoding or file compression. To this software, we apply our tool, either manually identifying functions as potential accelerator entry points, or leaving the tool unguided.

Our tool then responds, for each selected function, how much data moved into and out of it (and its callees), and how much computation it (and its callees) performed. Here, we assume that we will accelerate an entire function and all its callees (i.e., the callgraph below the function).

By considering these numbers in the limit, we determine whether a particular accelerator will be memory-bound, suggesting careful attention should be paid to its interface to the memory system, or whether it will be computation-bound, suggesting the need for more computational parallelism.

III. Applying our Methodology to a Simple Application: Image Rotation

We will use a simple task—rotating a square image 90 degrees clockwise—to illustrate the operation and use of our pintool. By design, this example is simple enough so that the results are easy to predict; this exercise is purely to illustrate the operation of our tool.

Figure 2 shows the code: a C program that reads a PPM image file into memory using `read_ppm()`, which repeatedly calls `getchar()`, rotates it in place using one of two algorithms, and then writes it to a file using `write_ppm()`, which uses `putchar()`.

A central goal of our pintool is to collect information that depends primarily on the *task* to be performed, and not the *algorithm* used to accomplish it, because any reasonable accelerator will employ at least a slightly different algorithm. Since we want to ask what-if questions, we want to divorce ourselves from our starting point without losing sight of the tasks we ultimately need our accelerators to perform.

To illustrate our tool's ability to be algorithm-agnostic, our example uses two different algorithms to perform rotation.

1) The first, `iter_rot()`, is a simple, iterative algorithm that iterates over the pixels in one quadrant of the image, shifting the four pixels in the corresponding position in each quadrant to the next quadrant.
2) The second, `rec_rot()`, divides the image into four quadrants, translates each of them in a clockwise direction, then calls itself recursively on each quadrant. This unusual algorithm, which we took from Goldberg and Robson [2, p. 408], only needs to perform block translation operations ("bitblts"), which may be easier to accelerate because they always access memory sequentially.

A. Checking For Any Dependence on Problem Size

Our pintool analyzes dynamic program behavior, so a preliminary question is whether its behavior depends strongly on the input. To answer this, the tool provides a breakdown of computation (light) and communication (shaded) for each run.

Figure 3 shows the breakdown of computation and communication for both the iterative and recursive rotate algorithms. We ran it on a range of image sizes (from 8×8 to 512×512 pixels). While the total work increases roughly proportionately to the image size, Figure 3 indicates that the relative ratio of computation to communication is nearly constant. The tiny

```
#include <stdio.h>
#include <stdlib.h>
#define PIX(x,y) raster[(x) + (y)*wd]
unsigned wd, ht, maxval, *raster;

void rec_rot(int x, int y, int s) {
  int i, j;
  s >>= 1;
  for (i = 0 ; i < s ; ++i)
    for (j = 0 ; j < s ; ++j) {
      int rgb = PIX(x+i, y+j);
      PIX(x+i,   y+j  ) = PIX(x+i,   y+j+s);
      PIX(x+i,   y+j+s) = PIX(x+i+s, y+j+s);
      PIX(x+i+s, y+j+s) = PIX(x+i+s, y+j);
      PIX(x+i+s, y+j  ) = rgb;
    }
  if (s <= 1) return;

  rec_rot(x,y+s,s);
  rec_rot(x+s,y+s,s);
  rec_rot(x+s,y,s);
  rec_rot(x,y,s);
}

void iter_rot() {
  int x, y, s;
  s = wd >> 1;
  for (y = 0 ; y < s ; ++y)
    for (x = 0 ; x < s ; ++x) {
      int rgb = PIX(x, y);
      PIX(x,      y     ) = PIX(y,      ht-x-1);
      PIX(y,      ht-x-1) = PIX(wd-x-1, ht-y-1);
      PIX(wd-x-1, ht-y-1) = PIX(wd-y-1, x     );
      PIX(wd-y-1, x     ) = rgb;
    }
}

void read_ppm() {
  int x, y;
  scanf("P6 %d %d %d ", &wd, &ht, &maxval);
  raster = (unsigned *)malloc(wd * ht * sizeof(int));
  for (y = 0 ; y < ht ; ++y)
    for (x = 0 ; x < wd ; ++x) {
      int rgb = getchar() << 16;
      rgb |= getchar() << 8;
      rgb |= getchar();
      PIX(x,y) = rgb;
    }
}

void write_ppm() {
  int x, y;
  printf("P6\n%d %d\n%d\n", wd, ht, maxval);
  for (y = 0 ; y < ht ; ++y)
    for (x = 0 ; x < wd ; ++x) {
      int rgb = PIX(x,y);
      putchar(rgb >> 16);
      putchar((rgb >> 8) & 0xff);
      putchar(rgb & 0xff);
    }
}

int main(int argc, char** argv) {
  if (argc != 2 || (argv[1][0] != 'r' && argv[1][0] != 'i')) {
    printf("USAGE: rotate [ir]\n"), exit(0);

  read_ppm();
  if (argv[1][0] == 'r') rec_rot(0, 0, wd);
  else                   iter_rot();
  write_ppm();

  return 0;
}
```

Fig. 2. An image rotation program. This reads a square PPM file and rotates it by 90° using either a recursive or an iterative algorithm. We chose rotation because of its simplicity and the multiplicity of available algorithms.

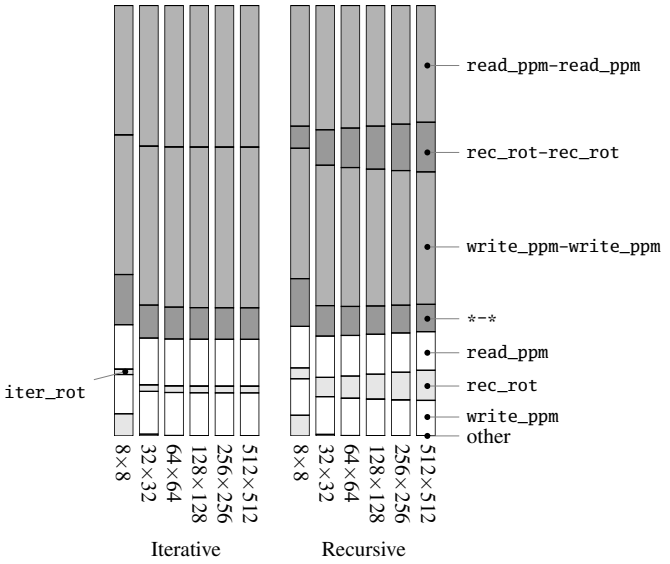


Fig. 3. Fraction of communication and computation for various image sizes for the iterative and recursive rotation algorithms. For all but the smallest image size (8×8), the breakdown varies little with image size.

8×8 image is one exception, where the fixed startup overhead accounts for a larger relative proportion of runtime. The other exception is the computation time for the recursive rotate function, which grows slowly with image size. This is due to the overhead of the recursive calls, which grows as $O(\log n)$.

The final observation is that `getchar()` and `putchar()` I/O operations dominate the computational work time.

Examining the communication patterns, we see, in all cases, that the total volume of data transfers in the application are dominated by local communication (i.e., data transfers within the `read_ppm()` routine). It is clear from this data that the overall behavior and nature of this application does not vary with input problem size. We can thus proceed to further examine a single input (128×128 pixels) assured that it is representative of all input sizes.

B. Locating Hotspots of Computation and Communication

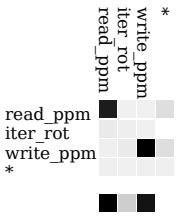
Having established that the behavior of this application does not change appreciably with the size of the input data (i.e., everything is $O(n)$), the next question is which functions are most costly and should therefore be accelerated.

To help answer this, our tool reports computation costs for each function and communication costs between each pair of functions. Figure 4 shows this data for the iterative and recursive rotation algorithm.

In this case, the general features of the instruction count data are somewhat intuitive based on a cursory review of the source code. What is significantly less obvious is the data movement through the application. In the next section we present the pintool that was designed to gather and reveal this information about an application's precise data usage patterns. Figure 4 shows the pintool's numerical and graphical output regarding computation and communication hotspots when run on the iterative and recursive rotations of a 128×128 pixel image.

Iterative Algorithm

	`read_ppm`	`iter_rot`	`write_ppm`	*other*
`read_ppm`	607413	0	0	49172
`iter_rot`	16387	4101	0	1
`write_ppm`	5	16384	689163	49155
other	4	0	18	491
Computation	804340	114820	738584	1902



Recursive Algorithm

	`read_ppm`	`rec_rot`	`write_ppm`	*other*
`read_ppm`	607413	0	0	49172
`rec_rot`	27306	216888	0	4
`write_ppm`	5	16384	689163	49155
other	5	0	18	491
Computation	804340	503549	738584	1902

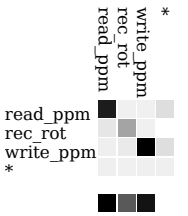


Fig. 4. Data from our pintool indicating the communication and computational hotspots for the two rotation algorithms run on a 128×128 image. Communication numbers count bytes transferred; computation numbers count arithmetic and logical instructions.

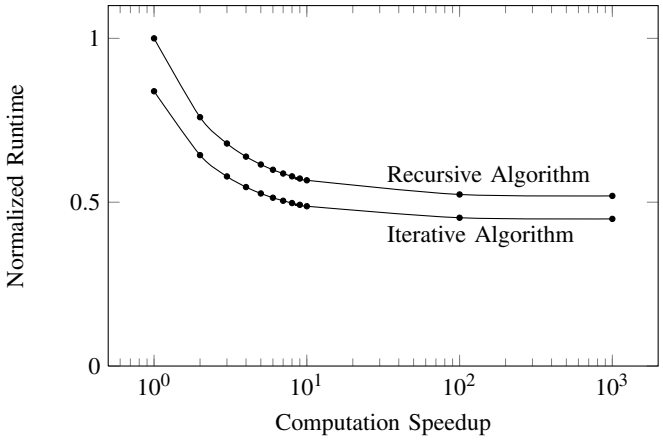


Fig. 5. The memory wall, as seen in the recursive and iterative image rotation kernels. As computation is accelerated, communication speed begins to dominate, limiting the maximum speed-up.

C. Hitting the Memory Wall

We first use this model to illustrate the well-known, well-understand phenomenon of the memory wall. The memory wall appears as the processing rate of the CPU accelerates increasingly much beyond the processing rate of the memory. Running the two versions of our rotation algorithm, accelerating the computation while leaving the memory access time fixed, we see in Figure 5 that performance is very quickly bound by the memory (i.e. communication) speed.

D. Distinguishing Local from Global Communication

From what we know of our example application, and what we know of the memory wall, it is clear that we need to accelerate not only computation, but communication as well. There are really two classes of communication. There is "accelerator local" communication which is communication that is entirely internal to an accelerator, and then there

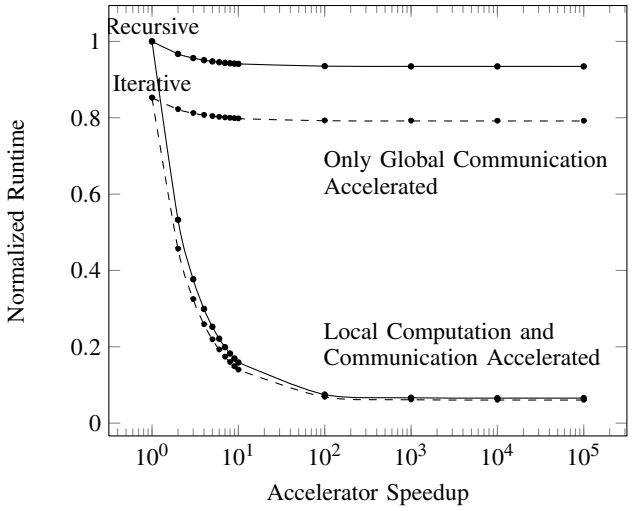


Fig. 6. Comparative speedups for image rotation when accelerating the global environment versus accelerating the the local accelerators

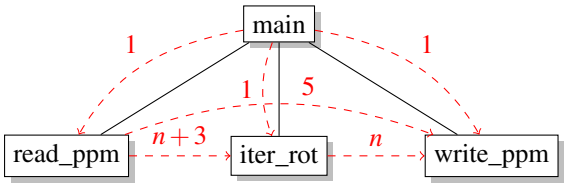


Fig. 7. Dataflow counts in words for the iterative rotate algorithm. n is the number of pixels in the image.

is "global" communication between accelerators. Figure 6 shows the relative benefit of accelerating local computation and communication versus focusing on accelerating the global communication system for the two rotation algorithms. As is quite clear the biggest speedups, for both algorithms, are to be found accelerating local communication.

We can also observe in Figure 6 how accelerating both the computation and the internal communication of a kernel fully decouples the kernel from the external system. Accelerating each kernel to the limit ultimately erases any difference between them, leaving the overall performance limited by the environment and the kernel's external interfaces.

Demonstrating potential speedups is one thing; realizing them is something else. However, accelerating local communication has signs of feasibility. In local communication, we have communication that is specific to a single task. A natural approach is to take advantage of specialization and to accelerate a kernel's internal communication in addition to its computation. This can be accomplished via a kernel-specific accelerator-local store to the accelerator. This local store provides tailored, and thus faster, communication resources for data transfers that are local to the algorithm.

E. Understanding Detailed Dataflow Behavior

Even after we have decided what functions to accelerate, our tool assists in the detailed design of such accelerators by providing insight into the way data is passed around. Figure 7

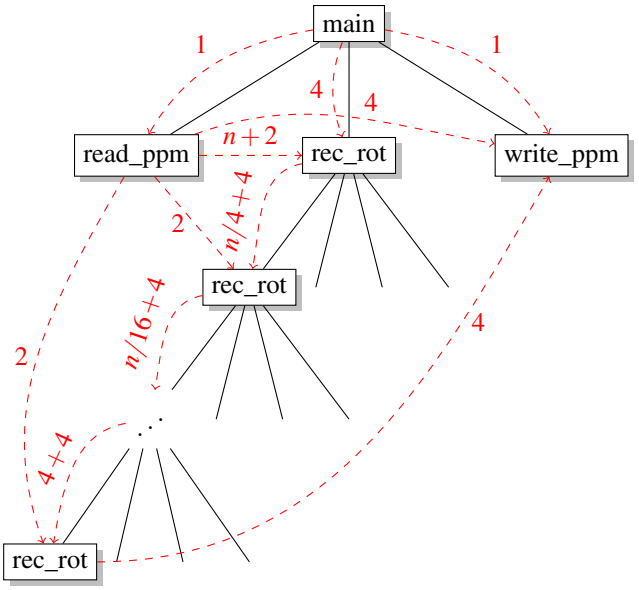


Fig. 8. Dataflow counts in words for the recursive rotate algorithm. n is the number of pixels in the image.

and Figure 8 show the detailed, parametric dataflow for the two rotation algorithms superimposed on the call trees. To derive these, we used data from our tool from multiple runs and manually massaged it make it easier to view.

The dataflow in the iterative algorithm is fairly straightforward. The `read_ppm` function transforms the image data into one word per pixel, which the `iter_rot` function reads, and finally `write_ppm` writes it. The extra data are the function return address (`main` to each of its children); the width of the image, the base address of the image, and `read_ppm` to `iter_rot`; and the height, width, maxval, image base, and `read_ppm` to `write_ppm`.

The dataflow in the recursive algorithm is a little more subtle. `rec_rot` calls itself recursively; each invocation works on rotating its area, subdivides the area into four, and passes this to four child calls. The recursion terminates when `rec_rot` attempts to rotate a 4×4 tile.

Only the leaf invocations of `rec_rot`, which are essentially tail-recursive, directly pass data to the `write_ppm` function. Ultimately, one word per pixel in the image is transferred; only one representative is shown in Figure 8.

IV. Our Pintool

Our pintool collects an annotated invocation tree representing the execution of the instrumented program, which we used to collect the data for the rotation example in Section III. Below, we describe the operation of our tool in detail.

Our tool produces four log files, which contain information about functions, the call tree, instruction counts, and memory transfer statistics. Conceptually, we instrument each instruction with an action for each log, although some of the actions are vacuous.

The running tool maintains a call stack, which records a unique ID for each function invocation and its frame pointer, which we use to match up calls and returns. We instrument calls and returns with code that maintains this stack; other parts of the tool use it to determine which function is running.

- **The function log** records the function name for each invocation. The other logs refer exclusively to invocation IDs, which this log makes human-readable. We write an entry to this when a call instruction executes. This log is produced in text or binary format using `-tfunction` and `-bfunction` flags respectively.
- **The subcalls log** records the call tree of the application as a list of sub-invocation IDs for each function invocation. The `-tsubcalls` and `-bsubcalls` flags generate this log in either text or binary form.

  To generate this, our tool maintains a hash from from live function invocations to a list of children, to which we append when a call executes. At each return, we write the invocation list of the returning function to the log and remove it from the hash table.
- **The instruction count log** records the number of non-memory-read/write instructions executed during each invocation. These counts can be used to identify execution hotspots for acceleration. This log will be created using either the `-ticount` and `-bicount` flags.

  To count instructions, we instrument each basic block with code that increments an entry in a hash table mapping function invocations to a count. We do not count data transfer or control-transfer instructions, only arithmetic and logical operations because we believe these are more representative of what an accelerator would have to do. Data transfers are more a side-effect of compiling for the x86, arising from its very limited number of registers, and control transfer operations are part of the control, which almost always can be improved upon in an accelerator.

  Upon a procedure return, the final instruction count for the terminating procedure is written to the log file and its entry is removed from the hash table.
- **The data transfer log** is the most interesting: it records the number of bytes transferred into each function invocation, broken down by source function invocation (i.e., who wrote the bytes). This can be thought of a list of weighted, directed edge between nodes in the invocation tree. This log is produced if either the `-txfers` or `-bxfers` flag is given. The granularity can be changed from words to bytes using the `-xfer-chunk` flag which sets the size, in bytes, of the blocks of memory to be tracked.

  To compute these statistics, our tool maintains a hash mapping memory locations to the invocation IDs of the last reader and last writer. When we encounter a memory write, we set its last writer to the invocation ID of the current function and clear the last reader ID. At a read, we check this hash table. If the current invocation ID is the last reader, we ignore the read because it represents the same invocation re-reading data it already learned about. Otherwise, we credit the invocation with a read from the last writer and update the last reader ID to the current function invocation.

  To report transfer statistics, we maintain a hash from invocation ID pairs (source and destination) to transfer counts. The contents of this hash is updated at read operations as described above, and it is written, in part, at every function return. There, we log the number of bytes transferred into the function and remove these entries from the hash.

  We only record read operations because they necessarily imply a matching write. This deliberately does not count data that is written and never read; we assume a shrewd accelerator designer would identify and optimize away such behavior.

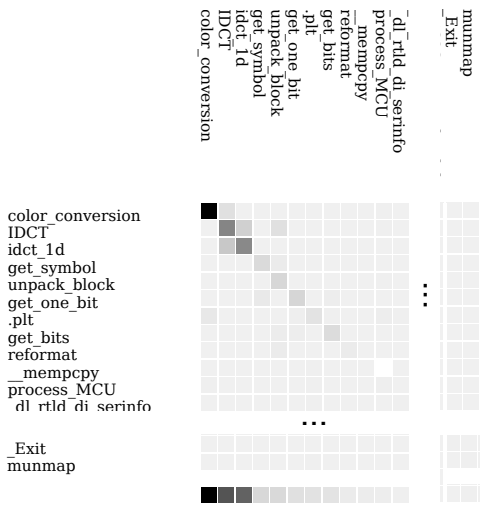


Fig. 9. The busiest functions, in terms of communication (top matrix) and computation (bottom row), in the JPEG decoder. The functions are ranked by their total amount of computation. We only show the top 10 functions.

V. Exploring JPEG Acceleration

We now demonstrate our pintool on a more complex application: JPEG decoding, which we ran on a 40K image.

We started by gathering some information about the application: we used the data from the pintool to generate the "heatmap" shown in Figure 9. Because this example has over 100 unique functions, we cropped the image for legibility. The functions are sorted by overall computational load. From prior knowledge about JPEG encoding, we expected IDCT to dominate, and indeed it does rank highly in terms of total computation. However, we were surprised to find color conversion ranking as highly as it did. This feature of the application we discovered only once the pintool had revealed the implicit computation and memory usage.

To understand the potential performance improvement of these hotspots, we selected the top four functions, `color_conversion`, `IDCT`, `idct_1d`, and `get_symbol`. Accelerating each one individually, we found the speedups shown

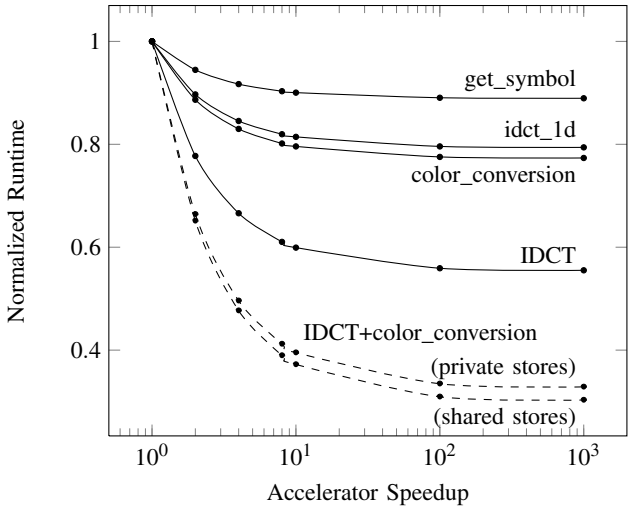


Fig. 10. The potential speedup from accelerating one or more functions

in Figure 10 (—●—). Though these are the hottest spots, we find we are still fighting Amdahl's law with no single function yielding large overall speedups. The fact that the speedups are small for individual functions indicates a reasonably well-balanced starting application.

When no single accelerator produces satisfactory speedups, a natural next step is to consider accelerating multiple functions. In this case we have selected the two most effective accelerators, `IDCT` and `color_conversion`, and evaluate two hardware configurations which accelerate both (Figure 10, -●-). In the first configuration with private accelerator stores the two accelerators each have their own individual accelerator store. In the second configuration with a shared accelerator store, the two accelerators share a single, store. From the data in Figure 10 we learn that, in the limit, the "shared" store architecture is only marginally better than the "private store" arrangement. Such insights can be important in effectively steering the accelerator design process.

VI. Related Work

A number of researchers have used dynamic binary instrumentation for related, but not identical, applications.

Nethercote and Seward [5] describe how Memcheck (part of Valgrind) tracks all the memory used by a program for the purposes of checking for errors like reading uninitialized values and array bounds violations. It actually tracks state for each *bit* of memory to identify problems with, say, bit-field operations. We take a more naïve approach.

The Redux system of Nethercote and Mycroft [4] is more ambitious than our tool: it aims to collect complete dataflow information about information passed around in a running program, including data in registers. Our focus is just on memory because we assume data passed through registers is part of the computation that will be accelerated and thus we do not considering it a limiting factor.

Our work bears a slight resemblance to taint analysis, which has been implemented using dynamic binary instrumentation, among other techniques. Like our application, the goal of taint analysis is to understand program dataflow, but for the purpose of identifying potential security risks. The system of Clause et al. [1] builds on Pin and also looks at dataflow within functions, but their analysis also chooses to take control dependencies into account, something we do not need to do.

Olszwski et al. [6] uses dynamic binary instrumentation to implement software transactional memory. Their system intercepts and records memory accesses within atomic regions, then dynamically adds code that commits the operations to memory. Our desire to understand memory access behavior is similar, but their ultimate objective is very different.

VII. Conclusions

In this paper we presented a pintool that tracks implicit computational and communication load across the functions of an application. We have demonstrated its exploratory power on the small example of image rotation and on a larger more obfuscated JPEG decode computation. Put together, the pintool and limit-based accelerator evaluation methodology employed here, allows designers and architects to explore many critical questions relating to accelerators:

- How should one evaluate a kernel as a candidate for acceleration?
- When designing an accelerator for a particular kernel, where should one's efforts focus, on computation or local storage?
- Does the "correct" answer to the preceding questions depend on the architecture of the larger chip (e.g., the global communication infrastructure or the presence of other on-chip accelerators)?

These are the questions the community will need to answer as we investigate accelerator-based chips. We believe that the data collection tool and modeling methodology presented in this work will be an asset in those investigations.

References

[1] James Clause, Wanchun Li, and Ro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 196–206, London, England, July 2007.

[2] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[3] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of Program Language Design and Implementation (PLDI)*, pages 190–200, Chicago, Illinois, June 2005.

[4] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2):149–170, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[5] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual Execution Environments (VEE)*, pages 65–74, San Diego, California, June 2007.

[6] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 365–375, Brasov, Romania, September 2007.