Accelerating Dynamic Binary Translation with GPUs
Chung Hwan Kim, Srikanth Manikarnike, Vaibhav Sharma, Eric Eide, Robert Ricci
School of Computing, University of Utah
{chunghwn,smanikar,vaibhavs,eeide,ricci}@utah.edu

Abstract

Binary translation is the emulation of one instruction set by another through translation of code. In binary translation, sequences of instructions are translated from the source to the target instruction set. Dynamic binary translation (DBT) looks at a short sequence of code – typically on the order of a single basic block – then translate it. Code is only translated as it is discovered and when possible, and jump instructions are made to point to already translated and saved code. In this project, in addition to the existing sequential translator, we propose to use a supplementary translator implemented on GPUs to accelerate DBT.

1. Introduction

In contrast to static binary translation, DBT has relatively low startup overhead since it translates code as needed. Because only a small set of basic blocks (hot working set) is executed frequently for most programs, existing translators maintain a code cache for translated instructions, and try to optimize the layout of the cache to minimize execution time. However, a significant amount of time is required before this optimized cache layout is reached. Moreover, when used in a broader range of applications, translator's performance cannot rely on the presence of hot working set. Dynamic binary translation is widely exploited in the areas of performance analysis, program optimization, security enhancement, and hardware virtualization.

2. Suitability for GPUs

There is an Inherent Parallelism in DBT,

DBT exhibits
• Inter-block and
• Intra-block parallelism

Blocks do not contain dependencies at the time of translation. Additionally, each block can be fragmented into independent instructions, each fetched to individual cores.

Additional Features
Supplementing the sequential translator with a GPU translator adds additional overhead on the CPU. Sequential code must be modified to

• Check level-2 cache for presence of required blocks
• Copy blocks from level-2 cache to level-1 cache

Additionally, we create a new translator model to

• Indicate start and number of blocks to be translated on GPU
• Send and receive blocks to and from GPU
• Ship blocks to level-2 cache

Measurement Model

To measure performance improvement, we have designed a mathematical model to compare the time taken for DBT with and without GPUs.

For DBT on a CPU, total time for translation and execution for current block, \( \tau \) can be estimated as

\[
\tau = c_1 * E + (1 - c_1) * (T + E)
\]

Where,

- \( c_1 \) – hit rate in level-1 code cache
- \( E \) – Time taken for CPU to execute a block
- \( T \) – Time taken for CPU to translate a block
For DBT with GPUs,

$$\tau' = c_1 E + (1-c_1) \times [(1-c_2) \times (T + E) + c_2 (\alpha + E)]$$

Where,
- $c_2$ – hit rate in level-2 code cache
- $\alpha$ – Time taken by CPU to copy a block from level-2 to level-1 code cache

$\tau'$ does not include a factor specific to GPU computation as the cache hit rates, $c_1$ and $c_2$, indicate the layout and availability of a block after GPU translation. The CPU and GPU translators work concurrently. The existing model of sequential DBT makes use of one process for all its operations and does not utilize the computation power of a multi-core system. Presence of GPU translator does not hamper the execution of the sequential translator as additional processes can be executed on different cores.

**Performance Analysis**

In this section we describe three scenarios for categorizing the time taken to translate a block, in comparison to the sequential model.

**Case 1:** When $c_1 = 0$ and $c_2 = 0$ (i.e. no cache hits to level-1 and level-2 caches),

$\tau' < \tau$

All blocks not present in either of the caches have to be translated. Since GPUs translate at a larger rate than a CPU, number of blocks in this case linearly increases the efficiency of the system.

**Case 2:** When $c_1 = 0$ and $c_2 = 1$ (i.e. no cache hits to level-1 cache whereas, all hits to level-2 cache),

$\tau' \geq \tau$

This is the most common case for DBT with GPUs as most blocks fall under this category. A significant number of blocks required to be copied from level-2 to level-1 cache and then executed. The crucial part in this case lies in how quickly the blocks are copied as compared to the time taken to translate the block. We believe that this is the case that decides using GPUs can make significant contributions to accelerate DBT.

**Case 3:** When $c_1 = 1$ (i.e. all hits to level-1 cache),

$\tau' = \tau$

Once a convergence point has been attained where the entire hot working set is laid out in the level-1 code cache in an optimized fashion, all that the CPU needs to do is execute the blocks.

3. How DBT works

Existing binary translators, such as Qemu and HDTrans run as following.

**Loop:**
- Form a block
  - if < block-miss in cache >
  - Translate block
  - Ship block to cache
  - Fetch block to CPU

**Figure 2:** Main procedure of existing binary translators

Existing DBT gathers a fixed number of instructions (about 15-17 instructions) to form a basic block to translate. However, when a jump-like instruction (RET, JMP, JCC, etc.) is reached, it stops gathering instructions and forms a block immediately. Next, it searches the code cache to find out whether the block has already been translated and stacked in the cache. If the search succeeds, it fetches the block directly to the CPU for execution. If the code cache does not have the block, it translates the block and puts it in the cache for later use.

In contrast, the following indicates the operations of our new model of DBT with GPUs.
As mentioned earlier, we have two different procedures that run simultaneously. We slightly modify the procedure of existing binary translators to support two-level code cache and call it CPU translator. CPU translator does both translation and execution whereas GPU translator only does translation.

The procedure of GPU translator is simple. It decides which blocks and how many blocks to translate at each iteration based on a heuristic and lets the device initiate parallel translation by sending a chunk of instructions. After the translation is complete, it receives the translated instructions from the device and ships the blocks to level-2 cache.

CPU and GPU translators can either be implemented as threads or as processes to work concurrently. However, we choose a multi-thread model to reduce the communication overhead between the two translators. The two threads can share memory for both level-1 cache and level-2 cache; thereby the only communication overhead becomes memory copy.

4. Words about HDTrans and CUDA

We picked HDTrans as it was one of the best translation technologies available in the market, but little did we know the issues with HDTrans. HDTrans is not a typical translator, but it’s more of a binary instrumentation tool. It first decodes and then translates the instructions. And this happens in a single loop and to make things even worse the two steps are closely knitted which made it very hard for us to break the loop. We started with trying to add CUDA support with the HDTrans but a point came when we realized that it will be impossible to do so because of the inherited complexity of the code.

4.1. Simplifying HDTrans

We decided to write a new translator, based on HDTrans, structured in a way to facilitate the CUDA implementation. Rather than reading the binary file directly we created an array of bytes from where we

---

**Figure 3: Main procedures of the new DBT model with GPUs**

As mentioned earlier, we have two different procedures that run simultaneously. We slightly modify the procedure of existing binary translators to support two-level code cache and call it CPU translator. CPU translator does both translation and execution whereas GPU translator only does translation.

**Figure 4: Two-level code cache: level-1 cache and level-2 cache**

CPU translator forms a basic block at the beginning of each iteration and checks if its code cache has the block already translated. Unlike the existing DBT, the new DBT searches through the level-2 code cache when there is a cache-miss in the level-1 code cache. If it finds the block in level-2 cache, it copies the block to level-1 cache. It exploits the same routine that existing DBT uses for shipping blocks. Here, an important post-process is needed. Existing DBT changes the operands (addresses) of jmp instructions during translation, to allow basic blocks to be directly fetched to CPU. However, the GPU has to find out if the block that the jump instruction is pointing to resides in level-1 cache. It is difficult for the GPU to do this. Our strategy to solve this problem is to take this address update to a post-process after shipping the block from level-2 cache to level-1 cache. If the level-2 cache does not have the block, it finally translates the block on its own, and transfers it to level-1 cache. The rest of the process is fetching the block to the CPU.

**Figure 5: Speculative Translation of Blocks**

The procedure of GPU translator is simple. It decides which blocks and how many blocks to translate at each iteration based on a heuristic and lets the device initiate parallel translation by sending a chunk of instructions. After the translation is complete, it receives the translated instructions from the device and ships the blocks to level-2 cache.
read the instructions. This helped us to work just on the translation of instruction rather than decoding them. Only during the decode process the exact length of instructions is calculated and in our scenario we need to know the instruction length beforehand. For this we assumed the length of instruction be 1 Byte. By taking this step we ensured an easier CUDA implementation.

4.2. Implementation on CUDA

The goal of this project was to show that GPUs can be used for the parallelization of binary translation. Once we had a simpler version of HDTrans at our disposal, the next step was to write the CUDA equivalent of the code. To run the translator in GPU we had to copy the lookup table into the GPU memory, once we had the lookup table in the memory we just modified the CPU code to make it run on the GPU.

5. Plan

We divided the project into three major milestones – feasibility experiments, design and porting of HDTrans on GPU. These milestones were arrived at keeping in mind course deadlines such as Project Proposal, Design Review and Poster Presentation and Demo. The team met every Monday to sync up on the activities performed over the week. We discussed and analyzed possible roadblocks each of us were facing and what direction to proceed in.

Vaibhav took ownership of conducting the feasibility experiments while ChungHwan and Srikanth worked on analyzing the main translate loop of HDTrans to identify the portions of the code that need to be ported onto the GPU and understanding the decode function of the binary instrumentation tool respectively.

We experimented with the feasibility of project by conducting two experiments - a) comparison of time overhead for memory copy from level-2 code cache to level-1 code cache and b) time allotted by the scheduler to a process with multiple threads as compared to that allocated to multiple processes. This experiment, however, was later opted out of the design and hence its results shall not be discussed in the next section.

A significant amount of time had to be spent on reworking the design of our translator owing to the design issues of HDTrans (Section 4). We also spent about 20% of the total project time on testing and collecting performance numbers.

6. Evaluation

The evaluation of this project was conducted on a commodity laptop (net-book) equipped with a CPU and GPU. The CPU was an Intel Atom N450, 1 core (hyper-threading) processor with a clock rate of 1.66 GHz and cache size of 512 KB. The GPU was an NVIDIA ION2 with a clock rate of 1.09 GHz with 16 cores (2 SMs x 8 SPs) capable of running CUDA 1.2 with 512MB global memory, 16KB shared memory and 16K registers per block.

The CPU is designed for energy efficiency hence it has almost the worst computing power amongst the modern CPUs. The GPU is also one of the cheapest models that support CUDA.

We measured and compared the performance of our CUDA implementation with the CPU implementation. We used variety of size of input to analyze the performance. All the time mentioned in the tables is in CPU ticks.

<table>
<thead>
<tr>
<th>Size</th>
<th>CPU time</th>
<th>GPU w/ memory copy</th>
<th>GPU w/o memory copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>512 B</td>
<td>460820</td>
<td>292610</td>
<td>169180</td>
</tr>
<tr>
<td>1KB</td>
<td>784160</td>
<td>300140</td>
<td>161420</td>
</tr>
<tr>
<td>2KB</td>
<td>1152210</td>
<td>345040</td>
<td>189230</td>
</tr>
<tr>
<td>4KB</td>
<td>2143790</td>
<td>401510</td>
<td>223200</td>
</tr>
<tr>
<td>8KB</td>
<td>314550</td>
<td>489240</td>
<td>265370</td>
</tr>
<tr>
<td>16KB</td>
<td>7199820</td>
<td>783740</td>
<td>454590</td>
</tr>
<tr>
<td>40KB</td>
<td>13257760</td>
<td>1672240</td>
<td>1032210</td>
</tr>
<tr>
<td>80KB</td>
<td>2486300</td>
<td>3283190</td>
<td>1297690</td>
</tr>
<tr>
<td>1MB</td>
<td>313432980</td>
<td>37780400</td>
<td>14798510</td>
</tr>
<tr>
<td>2MB</td>
<td>628973410</td>
<td>70816780</td>
<td>29451030</td>
</tr>
</tbody>
</table>

Table 1: Evaluation Results

Figure 7: Performance evaluation graph indicating the change in CPU/GPU Time in ticks against change in size of code cache
As we need to copy the translation tables to the GPU memory before the translations starts, our CUDA implementation has an overhead of 1,401,634,280 CPU cycles.

7. Challenges

Adding a supplementary GPU translator to the CPU involves a lot of challenges. Although GPUs have high computation power they have equally high overhead for data transmission. One of our challenges would be to fine-tune the number of blocks and the timing of the translation on GPUs to achieve performance improvement.

Currently, our DBT model aims at the same source and target architecture, more specifically IA-32. Although, translating from one architecture to the same is widely used in many areas, another challenge we perceive is implementing DBT for different source and target architectures.

Though we are using a large secondary cache for storing blocks translated by GPU, we have to come up with a right cache line eviction strategy and also design a heuristic for control flow prediction to improve the performance of the whole system.

8. Conclusions and Future work

As dynamic binary translation is used widely in a variety of fields, any improvement in its performance will be of great benefit to the computing world. In this project, we try to accelerate DBT by supplementing the sequential translator with a finely tuned GPU based translator. We used a simplified Translator and ported it to CUDA. The one main problem which we need to work on is to how to find the size of instructions or at least have some heuristic to ensure that our system is able to translate instruction of all the sizes. We could use our system with RISC architecture as they have fixed length instructions but to do so we have to find some ARM machine with CUDA support.

As we want to convert this idea into a paper, our plan is to write our own fully functional implementation of translator aimed towards facilitating CUDA porting. We also have few heuristics in our mind to detect the size of instructions which we would test to ensure a complete translator implementation.

9. References
