Hardware/Software Helper Thread Prefetching On Heterogeneous Many Cores

Bharath Narasimha Swamy\textsuperscript{1}, Alain Ketterlin\textsuperscript{2}, Andre Seznec\textsuperscript{1}

\textsuperscript{1} IRISA/INRIA, Rennes, France
\textsuperscript{2} Universite de Strasbourg & INRIA, Nancy, France
Heterogeneous Many Cores

- Mix cores of different sizes on die
  - few *large* cores - fast seq. execution
  - many *small* cores - power efficient parallel execution

- Why heterogeneity?
  - match application performance needs to profile of available cores.
  - migrate threads for energy efficiency.

- *legacy codes and difficult to parallelize applications cannot benefit from increased core count*
Using Helper Threads to improve Sequential Performance

Helper threads exploit unused hardware contexts to improve single thread performance

- helper code much smaller than main thread - target delinquent accesses, only execute pre-compute slice
- execute a reduced 'helper' ahead of time - provide perf. hints - trigger $ misses, for sequential thread running on large core
Improving Sequential Performance on Many Cores

This study

- Use small cores to execute helper threads - generate prefetch for memory intensive sequential threads on large core
- Prefetch to shared last level cache - applications that miss extensively in the L3 can benefit

Contributions
- Core-tethering: hardware/software framework for efficient execution of helper threads in HMCs, reduce overheads associated with initiation/control of helpers.
- Evaluate suitability of small cores to prefetch for a much powerful main core
Core-tethering

- Provides a co-processor like interface to the small cores
- Hardware support for spawning helper threads - large core can directly initiate and control helper execution on small cores
- Efficient transfer of execution context between the cores

**Helper Control Block Register (HCBR)** - helper action and the application context

**Helper Iteration Count Register (HICR)** - used for synchronization between main and helper.
Core-tethering

- New user mode instructions allow main core to initiate/control helper execution on small cores.

- **ISHCB** - initiate store helper control block
  - initiates a helper action, HCBR specifies required helper-action and (optionally) values of input registers required by helper
  
<table>
<thead>
<tr>
<th>Helper action</th>
<th>Contents of HCBR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPAWN</strong>: start execution of a helper codelet</td>
<td>instruction address of helper codelet + values of variables required as live-ins</td>
</tr>
<tr>
<td><strong>KILL</strong>: stop helper execution</td>
<td></td>
</tr>
<tr>
<td><strong>SYNC</strong>: synchronize execution of main and helper threads</td>
<td>values of variables required for synchronization</td>
</tr>
</tbody>
</table>

- **WRHCBR** - write helper control block register

- **RDHCBR** - read helper control block register
long refresh_potential (network_t *net) {

    /* INITIALIZATION */
    tmp = node = root->child;

    while (node != root) {
        while (node) {
            if (node->orientation == UP)
                node->potential = node->basic_arc->cost + node->pred->potential;
            else /* == DOWN */
                node->potential = node->pred->potential + node->basic_arc->cost;
            checksum++;
        }
        tmp = node;
        node = node->child;
    }
    node = tmp;
    while (node->pred) {
        tmp = node->sibling;
        if (tmp) {
            node = tmp;
            break;
        } else
            node = node->pred;
    }
    return checksum;
}
Core-tethering - illustration

```c
long refresh_potential (network_t *net) {
    /* INITIALIZATION */
    node_t *root = net->nodes;
    tmp = node = root->child;

    CT_CLEAR_HICR();
    CT_WRITE_HCBR( SPAWN, &refresh_potential_helper, root );
    CT_ISHCB();

    while (node != root) {
        /* TARGET LOOP */
        CT_SYNC_MAIN(node);
    }

    CT_WRITE_HCBR(KILL );
    CT_ISHCB();

    return checksum;
}

void refresh_potential_helper ( ) {
    node_t *node_pvt, *tmp_pvt;
    node_t *root;

    CT_CLEAR_HICR();
    CT_READ_HCBR(&root);

    tmp_pvt = node_pvt = root->child;

    while (node_pvt != root) {
        /* TARGET LOOP WITH PREFETCHES */
        CT_SYNC_HELPER(node_pvt);
    }

    CT_SYNC_HELPER (node_t * node_pvt) {
        INCR_HICR();
        if (HICR % LOOP_SYNC_INTERVAL == 0) {
            CT_READ_HCBR (&hicr_main, &node_main);
            if (HICR - hicr_main < MAX_DIST)
                HICR = hicr_main;
            node_pvt = node_main; //catch up with main
        }
    }

refresh_potential() in main thread
```

helper code for `refresh_potential()`
Evaluation

- profile information to identify target load/stores – helper constructed at the binary level
- trace driven simulation of large and small cores.
- modeled support for hardware based spawning of helper threads, low latency communication between main and helper cores
Evaluation

- **large cores (6-issue):** LC1 (128 entry ROB), LC2 (256 entry ROB)
  - high performance out-of-order superscalar processor, aggressive branch prediction, oracle data-dependence predictor, data prefetcher based on stream buffers.

- **small cores (2-issue):** SC1 (32 entry ROB), SC2 (64 entry ROB)

Comparison of Small and Large Cores (normalized to SC2)
### Evaluation

<table>
<thead>
<tr>
<th>Application name</th>
<th>Benchmark Suite, input</th>
<th>Helper</th>
<th># target loops</th>
<th>% inst covered by target loops</th>
<th>% inst in helper</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>NAS, B</td>
<td></td>
<td>2</td>
<td>99</td>
<td>30.5</td>
</tr>
<tr>
<td>MG</td>
<td>NAS, B</td>
<td></td>
<td>1</td>
<td>99</td>
<td>6.4</td>
</tr>
<tr>
<td>lbnm</td>
<td>spec2006, ref</td>
<td></td>
<td>1</td>
<td>89</td>
<td>7.2</td>
</tr>
<tr>
<td>libquantum</td>
<td>spec2006, ref</td>
<td></td>
<td>2</td>
<td>98</td>
<td>24.5</td>
</tr>
<tr>
<td>mcf</td>
<td>spec2006, ref</td>
<td></td>
<td>2</td>
<td>59</td>
<td>55</td>
</tr>
<tr>
<td>milc</td>
<td>spec2006, ref</td>
<td></td>
<td>5</td>
<td>96</td>
<td>5</td>
</tr>
<tr>
<td>equake</td>
<td>spec2000, ref</td>
<td></td>
<td>3</td>
<td>90</td>
<td>10.2</td>
</tr>
<tr>
<td>radix</td>
<td>splash2x, native</td>
<td></td>
<td>1</td>
<td>92</td>
<td>11.5</td>
</tr>
<tr>
<td>svm-rfe</td>
<td>minebench</td>
<td></td>
<td>1</td>
<td>99</td>
<td>16.8</td>
</tr>
</tbody>
</table>

**Memory Intensive applications in the benchmark set**
Simulation Results (1)

- Small core (SC2) runs helper thread for large cores, LC1 and LC2

  - hardware prefetcher not effective for irregular access patterns
    helper prefetching : mcf improves by 1.12X(1.19X), much larger gains on radix

  - helper prefetching works synergistically with the hardware prefetcher for sequential access patterns

main core: LC1

main core: LC2
Simulation Results (2)

- Small core (SC1) runs helper thread for large cores, LC1 and LC2
- cg, equake and libquantum - sequential access pattern, but only partial prefetch conversion in helper thread

![Normalized Execution Time Graphs]

**main core: LC1**

**main core: LC2**
Simulation Results (3)

- Comparing helper thread on small Vs large cores (helper : SC2)
  - using SC2 within 7% of using LC1 (excepting libq, svm)
  - using SC2 within 19% of using LC2 (excepting libq, svm)

main core - LC1

main core - LC2
Simulation Results (4)

- Comparing (LC1 + helper prefetch) Vs LC2

- helper prefetching on small cores can provide an alternate design point for improving performance on memory intensive applications

![Normalized Execution Time](chart.png)

normalized to LC2
Conclusion

Evaluated helper thread prefetching in Heterogeneous Many Cores

- Helper threads on (moderately sized) small cores can significantly accelerate a larger core compared to using a hardware prefetcher alone.

- Small cores provide a good trade-off against using an equivalent large core to run helper threads in a HMC.

- On memory intensive applications, helper prefetching (on small cores) can provide an alternative to growing the instruction window size for achieving higher sequential performance.
Thank You
Backup – Helper Construction

- Helper construction techniques – delinquent selection and slice construction can be
  - statically using offline profile information, either at compile time or as a post-compilation directly on the binary
  - dynamically at runtime, either using special hardware support or as part of a software dynamic optimization system

This paper:

- statically constructed pre-compute slices from a compiler generated binary
  - static analysis on binary to build a Static Single Assignment (SSA) form of the program.
  - slicing starts on selected delinquent loads, and continues backward until reaching the boundaries of the region of interest (typically, a loop or a function).
  - use-def links that cross this boundary identify the set of live-ins, i.e. register values that are used as input to the helper.
Backup – Related Work

- Hardware techniques to overcome overheads of helper threading
  - [Hall et al.]: hardware based spawning of the helper thread using a branch to assist thread instruction that reduces thread-spawn overheads
  - [Gschwind et al.]: direct hardware-assisted communication between threads that bypasses the memory hierarchy for fast communication of values between the main and the helper threads,
  - [Lee et al.]: special synchronization registers in the memory processor running helper threads, to communicate synchronization variables between the main and helper with low overhead.
Backup – Related Work(2)

- Using a small core to accelerate sequential main thread
  - [Lau et al.]: partner cores pairs a larger compute core with a low area/power partner core that runs optimization code to trigger cache misses on behalf of the compute core.
  - [Lee et al.]: a loosely coupled simple core embedded in memory either in the DIMMs or the DRAM chips to run helper threads.
  - [Sohilin et al.]: a small processor placed close to the memory executes a software thread that performs memory side correlation prefetching.
  - [Woo and Lee]: COMPASS uses idle programmable shaders in a GPU to emulate prefetch algorithms in software and generate prefetches for a sequential application running on the CPU.