Performance Characterization of Parallel Game-tree Search Application Crafty

TAN Ying\textsuperscript{1,2}, LUO Ke-lu\textsuperscript{2}, CHEN Yu-rong\textsuperscript{1}, ZHANG Yi-min\textsuperscript{1}

\textsuperscript{1}. Intel China Research Center, Intel Corporation Beijing 100080 China
\textsuperscript{2}. School of Computer Science and Engineering, University of Electronic Science and Technology of China Chengdu 610054 China

Abstract  Game-tree search plays an important role in the field of Artificial Intelligence (AI). In this paper, we characterize one parallel game-tree search workload in chess: the latest version of Crafty, a state-of-art program, on two Intel Xeon\textsuperscript{®} shared-memory multiprocessor systems. Our analysis shows that Crafty is latency-sensitive and the hash-table and dynamic tree splitting used in Crafty cause large scalability penalties. They consume 35\%-50\% of the running time on the 4-way system. Furthermore, Crafty is not bandwidth-limited.

Key words  performance characterization; workload analysis; parallel game-tree search; computer chess; crafty

1 Application Characteristics

Crafty is a state-of-the-art chess program developed by Dr. Robert Hyatt, and is probably the strongest non-commercial chess program in the world freely distributed with source code. As in the latest release version 19.9, Crafty uses an alpha-beta-based parallel game-tree search framework, and more specifically, a similar Dynamic Tree Splitting (DTS) algorithm for its parallel implementation which is first introduced in Dr. Hyatt’s Ph.D. thesis\textsuperscript{[8]}.

The basic idea of this algorithm is to “split” the “move list nodes” (siblings) into several blocks, and whenever detect that one or more threads are in their idle loop, invoke this thread to begin a new alpha-beta search from the current node through copying the search state space for each thread working at this node, then sending everyone off to SearchSMP function to search this node’s block siblings. Fig.1 describes the DTS algorithm.

\begin{verbatim}
DTS(root) {
\textcolor{red}{\textbf{while} (Stopping\_criterion() == false) { \textcolor{blue}{
//One processor search to ply = N
SearchRoot(root); \textcolor{blue}{
//Detect free processors, and begin tree split
Split(node v); \textcolor{blue}{
//Initialize new threads.
ThreadInit(); \textcolor{blue}{
//Copy a “split block” to begin a new search
CopytoSMP(node v); \textcolor{blue}{
SearchSMP(node v); \textcolor{blue}{
}
ThreadStop(); \textcolor{blue}{
}
}\textcolor{red}{\textbf{}}} \textcolor{blue}{
}\textcolor{red}{\textbf{}}} \textcolor{blue}{
}\textcolor{red}{\textbf{}}} \textcolor{blue}{\textcolor{red}{\textbf{}}}
\end{verbatim}

Fig.1  The Crafty DTS algorithm

Search is one of the fundamental problems to Artificial Intelligence (AI). Over the decades, theoreticians and practitioners have excogitated tons of application-independent (branch and bound\textsuperscript{[1]}, the minimax algorithm\textsuperscript{[2]}, alpha-beta pruning\textsuperscript{[3]}, etc.) or application-dependent search methods, which are either generable to common computer world or high performances to specific domain problems. One important experimental hotbed of various search algorithms has been the field of game playing, in which game-tree search is widely used to find the best moves for two-player games\textsuperscript{[4-5]}.

Computer programs based on advanced search algorithms have achieved great success in popular games such as checkers, Othello and chess\textsuperscript{[6]}.

In this paper, we focus on characterizing one parallel game-tree search workload: the latest version of Crafty\textsuperscript{[7]}, a state-of-the-art chess-playing program, on two Intel Xeon\textsuperscript{®} shared-memory multiprocessor systems. We perform detailed micro-architectural analysis of the workload using the Intel performance tools such as Vtune. Previous work is mostly focused on researching various search strategies and their specific implementations using a variety of languages on different systems. To our knowledge, no attempt has been made to perform detailed performance characterization of the algorithms especially on the shared-memory multiprocessor system. Our analysis shows that Crafty is latency-sensitive, and the hash operations and dynamic tree splitting operations cause much larger penalties to the overall performance than we previously expected.
Crafty uses many enhancements, including transposition tables, killer-moves, null-move heuristics, etc. to the search algorithm\textsuperscript{[9-10]}. Among these enhancements, the transposition table (actually a large hash table) undoubtedly plays a very important role. The primary purpose of the table is to enable recognition of move transpositions leading to position (sub-tree) that has already been completely examined. In such a case, there is no need to search again\textsuperscript{[9]}. This technique has indeed great enhancement to single thread chess program, however, later experiments disclose that for multithreading program to maintain this table requires great synchronization costs.

2 Workload Construction

2.1 Software Environment and Hardware Platform

Our work is based on two Intel Xeon\textsuperscript{®} based shared-memory multiprocessor platforms. The first is a 4-way system referred to QP (Quad-Processors)-machine, and the second is a Unisys-Es7000 system which consists of 16 processors. Some key characteristics of these two systems are listed in Tab.1. We can see that the Unisys system has larger memory latency because the memory transactions need to go through L4 cache before visiting the memory. It may lead to less parallel speedup, since it is more difficult for the threads to hide memory latency for each other.

<table>
<thead>
<tr>
<th>Platform</th>
<th>QP-machine</th>
<th>Unisys-Es7000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>4-way Xeon MP 2.8 GHz</td>
<td>16-way Xeon MP 3.0 GHz</td>
</tr>
<tr>
<td>L1 cache</td>
<td>8 KB, hit latency 2 cycles</td>
<td>8 KB, hit latency 2 cycles</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512 KB, hit latency ~10 cycles</td>
<td>512 KB, hit latency ~10 cycles</td>
</tr>
<tr>
<td>L3 cache</td>
<td>2MB, hit latency 30+ cycles</td>
<td>4MB, hit latency 30+ cycles</td>
</tr>
<tr>
<td>L4 cache</td>
<td>None</td>
<td>32MB on-board, 300+ cycles</td>
</tr>
<tr>
<td>FSB/MHz</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>Interconnection</td>
<td>FSB</td>
<td>Crossbar</td>
</tr>
<tr>
<td>Memory</td>
<td>2 GB, peak bandwidth 2.1 GB/s</td>
<td>8 GB, peak bandwidth 3.2 GB/s</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.4.21 smp</td>
<td>Linux 2.4.21 smp</td>
</tr>
</tbody>
</table>

For software configuration, on both platforms, we use Intel C/C++ Compiler\textsuperscript{®} Version 8.1 to compile the two programs under Linux Kernel 2.4.20smp with full compiler optimization (-tpp7 –O3 -g). Specially, the “-g” option is to let the Intel Vtune\textsuperscript{®} Performances Analyzer carry out its source level instrumentation.

2.2 Test suite and program baseline

We use the Louguet Chess Test II (LCT-II), version 1.21 in our experiment\textsuperscript{[11]}. This test suite consists of 35 positions (the winning move is known), which can be divided into three main themes: positional, tactical and endgame. The test suite is usually used to estimate the rating of a chess program. We choose 2 positions out of each three position categories respectively, and make up of our 6 positions. Fig.2 shows the sequential running time of Crafty 19.9 of these 6 positions, and these results are also our baseline of the later experiments.

3 Workload Characterization

3.1 Experimental Methodology

We present several characteristics of Crafty workload obtained using the Intel Vtune\textsuperscript{®} Performance Analysis tools on our two hardware platforms. We also have a detailed analysis/comparison for function level of the both programs. For all experiments, we run Crafty for two iterations prior to taking measurements. This warm-up period allows the workload to warm up the data caches and enter into steady-state mode. All test datasets run for ten iterations during the measurement phase. All results are averaged and reported.

To obtain better understanding of the program’s behaviors, we also breakdown the program into function level using Intel\textsuperscript{®} Vtune tools, and divide them into 6 different classes, including the search operations, the move generation operations, the evaluation operations, the multi-threading overheads (including the threading costs, the lock/unlock critical
areas costs and operations of splitting tree costs), the hash operations (like initializing, loading and storing the hash table), and the other sequential operations. In this way, we can know which class of operations contributes most performance penalties to the overall program performance.

3.2 Scalability Analysis

Fig.3 and Fig.4 report the speedup curves of Crafty on both hardware platforms for 1, 2 and 4 processors. We can see that Crafty has better scalability performance on QP-machine than on the Unisys machine. On QP-machine, Crafty gets a steady increasing speedup and reaches an average speedup of 2.1X with 4 processors. However, on the Unisys machine, Crafty gets an overall decreasing speedup, which only gets an average speedup of 0.9X with 4 processors. In particular, Fin6 gets the highest speedup of 1.7X with 2 processors and also gets the modest speedup of only 0.6X using 4 processors.

It is surprising to see that hash operations take so much execution time, which is about 20% on QP-machine and 30% on the Unisys machine with 4 processors, respectively. This cost on the Unisys platform is much higher than that on QP-machine, because hash operations are mostly memory-access operations, and the hash table size is much bigger than the total cache size. So the Unisys machine with an added L4 cache has much more penalties for such operations. Furthermore, we can see that the dynamic tree splitting makes a lot of multithreading overheads. With up to 4 processors, it even reaches about 20%-30% on QP-machine and about 25%-50% on the Unisys machine.

3.3 Memory Hierarchy Analysis

3.3.1 Instruction Type Distribution

Fig.7 shows the average instructions executed (in billion instructions) and breakdown for Crafty for all datasets. We split the instructions into integers, loads and stores. There are none floating-point instructions in Crafty. We can see that Crafty is a memory access dominant program, in most cases the loads and stores instructions consist of about 50% of total instructions.
and the number of stores is about half of the number of loads.

As shown in Fig.3 and Fig.4, Crafty is more sensitive to memory hierarchy than to processor speed: although the processor speed is faster on Unisys-machine than that on the QP-machine, because Unisys-machine gets an extra L4 cache, this introduces more performance penalties. When running with single processor, the average running time is 25% slower and the instructions executed are 1.5 times more than that on QP-machine. When the number of processors increases, this performance gap between two platforms becomes larger: with 4 processors, the average running time is 1.9 times slower and the instructions executed are 2 times more than that on QP-machine. This is because Crafty spends more dynamic instructions for running the spin loops to solve the test positions while suffering more cache-misses or more synchronization. Further analysis confirms that the one hotspot spin-loop spends 28% of the Unisys dynamic instructions compared to 12% in QP-machine.

3.3.2 Cache Misses Behavior

Fig.8 shows the average L1, L2 cache miss rates per memory reference on both platforms. Crafty L1 cache miss rates are around 9%, and the L2 cache miss rates are around 4%.

Fig.9 shows L3 cache miss per thousand instructions for two programs on both hardware platforms. Because the L3 cache size on Unisys is 4MB, which is 2 times larger than on the QP-machine, it’s natural that the L3 cache miss will be smaller on Unisys, as a result, the average L3 miss rate for Crafty is about 3 times larger on QP-machine than on Unisys-machine. The L3 cache miss puts a critical impact on the overall program performance: take QP-machine for example, the average L3 miss latency is about 180 cycles[12]. As shown in Fig.9, there are about 1.666 L3 cache misses per thousand instructions for Crafty, as calculation blow, we hereby expect that for Crafty with 4 processors the L3 cache miss will bring 300 stall cycles miss latency for every 1000 instructions.

\[
\text{180 stall cycles} \times \frac{1.666 \text{ misses}}{1000 \text{ instructions}} \approx \frac{300 \text{ stall cycles}}{1000 \text{ instructions}}
\]

Further analysis confirms that the one hotspot spin-loop spends 28% of the Unisys dynamic instructions compared to 12% in QP-machine.
To give a more detailed performance characterization, we manage to give a functional breakdown to these two programs and see how the L3 cache misses are made up of Fig.10 and Fig.11 show the breakdown of total L3 cache miss rates for functional regions for Crafty on our two platforms. The hash operations contribute most of the L3 cache misses, sometimes, even over 97% on both platforms. When the number of processors increases, the cache size increases, then the hash operation misses for L3 cache decrease. However, in most cases, hash operations are still dominant contribution to the L3 cache misses. Due to the overall very high L3 cache miss rates we have observed in Fig.10, we can conclude that the hash operations are one of the major reasons influence the performance of Crafty.

3.3.3 Memory Bandwidth

Fig.12 and Fig.13 show the memory bandwidth usage Crafty on both platforms. As expected, the Front Side Bus (FSB) utilization rates of Crafty increase with the number of processors on both platforms. However, the bus remains under-utilized: even for 4 processors only less than 40% of the maximum available bandwidth is used on QP-machine for Crafty, respectively; and on the Unisys machine the FSB utilization rates is less than 20%. This observation suggests that Crafty is not bandwidth-limited.

4 Conclusions

In this paper, we characterize Crafty, one parallel game-tree search application in chess, on two shared-memory multiprocessor systems. We perform detailed micro-architectural analysis of this workload using the Intel performance tools. Our analysis shows that Crafty is latency-sensitive and the hash-table and dynamic tree splitting used in Crafty cause large scalability penalties. They consume 35%-50% of the running time on the 4-way system. Furthermore, Crafty is not bandwidth-limited.

References

Brief Introduction to Author(s)

TAN Ying (谭 胜) was born in Sichuan Province, China, in 1981. From 1999, he has been studying at the University of Electronic Science and Technology of China (UESTC) as a graduate student. From March 2005 to Oct. 2005, he joined the Intel China Research Center Micro-Processor Technology Lab as an intern. His research activities have been concerned with computer architecture, program optimization and parallelization, and data mining techniques.

LUO Ke-lu (罗克露) is a professor of School of Computer Science and Engineering, UESTC. She graduated in School of Computer Science of Chengdu College of Electronic Engineering in 1982 and was sent to Rome University of Italy by Chinese government for the advanced studies on computer technology in 1985. Her research interests including Real time system, CAD, etc.

CHEN Yu-rong (陈玉荣) is a researcher at Micro-processor Technology Lab, Beijing. Currently he conducts research on emerging computing paradigms, parallel algorithms, and scalable workload development and analysis. He joined Intel in 2004. Before that he spent two years doing postdoctoral research work in computer science in Institute of Software, Chinese Academy of Sciences. He received his B.S. degree in Applied Mathematics in 1998 and M.S. and Ph.D. degrees in computational mathematics in 2002, all from Tsinghua University, China.

ZHANG Yi-min (张益民) is researcher in Microprocessor Technology Lab, Beijing. He joined Intel in 2000. He received his B.A. degree from Fudan University in 1993, his M.S. degree from Shanghai Maritime University in 1996, and his Ph.D. degree from Shanghai Jiaotong University in 1999, all in Computer Science. He leads a team of researchers working on various statistical computing techniques and their scalability analysis.

(Continued from page 147)

Brief Introduction to Author(s)

ZHENG Dong (董 政) was born in Sichuan, China, in 1981. He received the B.E. degree in electrical engineering from University of Electronic Science and Technology of China (UESTC), in 2003. He is currently pursuing the Ph.D. degree with UESTC, and working at the Ministry Key Lab of Intelligent Mechatronics System in UESTC. The research interests include intelligent mechatronics system and intelligent fire detecting system.

HUANG Da-gui (黄大贵) was born in Guizhou, China, in 1952. He received the M.S. degree from UESTC in 1988. He is currently a professor and Ph.D. student supervisor with UESTC. Mr. Huang was awarded “The outstanding Chinese Master” by Sichuan Education Committee in 1991. He was honored “The Outstanding Midlife and Youth Expert” by China Ministry of Mechatronics in 1992. He has finished more than 20 scientific projects and has published more than 50 papers. His current research interests include intelligent mechatronics system, smart motion control, micro-electromechanical system and intelligent fire detecting system.

ZHANG De-yin (张德银) was born in Sichuan, China, in 1971. He received the B.E. and M.S. degrees from Nanjing University of Aeronautics and Astronautics in 1994 and 1997, respectively. He is currently pursuing the Ph.D. degree with UESTC. His main research activity is in the field of design and fabrication of intelligent UV and infrared fire sensors.