| LCOV - code coverage report |
| --- |
|  |
| | Current view: | [top level](http://docs.google.com/index.html) - [ArbitraryModeFlow/src](http://docs.google.com/index.html) - ArbitraryModeProcessor.cpp (source / [functions](http://docs.google.com/ArbitraryModeProcessor.cpp.func.html)) |  |  | Hit | Total | Coverage | | --- | --- | --- | --- | --- | --- | --- | | Test: | EBM9K Converter Coverage Report |  | Lines: | 132 | 164 | 80.5 % | | Date: | 2012-03-23 |  | Functions: | 11 | 11 | 100.0 % | | Legend: | Lines: hit not hit | Branches: + taken - not taken # not executed |  | Branches: | 31 | 76 | 40.8 % | |  | |
|  |

|  |
| --- |
| Branch data Line data Source code  1 : : #include "dec\_types.h"  2 : : #include "ArbitraryModeProcessor.h"  3 : : #include "print.h"  4 : : #include "Rect.h"  5 : : #include "SFTableCreator.h"  6 : : #include "VSBDataProcessor.h"  7 : : #include "SDDB.h"  8 : : #include "RetryFunc.h"  9 : :   10 : : namespace nft {  11 : :   12 : : ArbitraryModeProcessor::ArbitraryModeProcessor( ThreadID inThreadID,  13 : : OutputGenerator::Pointer inOutputGenerator )  14 : 46 : :Thread(),  15 : : mCellProcessors(),  16 : : mOutputGenerator(inOutputGenerator),  17 : : mMaxScheduledContainers(100),  18 : : mStop(),  19 : : mMutex(),  20 : : mWaitCondition(),  21 : : mNumActiveProcessors(),  22 : : mThreadDataSizes(),  23 : 46 : mDataProc() {  24 : 46 : setThreadID( inThreadID );  25 : : }  26 : :   27 : :   28 : 92 : ArbitraryModeProcessor::~ArbitraryModeProcessor() {  29 : 46 : nftFree(mThreadDataSizes);  30 [ + - ][ # # ]: 46 : delete mDataProc;  [ # # ]  31 : 46 : }  32 : :   33 : :   34 : : void  35 : 46 : ArbitraryModeProcessor::operator()() throw() {  36 : 46 : ConverterParams \*pParams = ConverterParams::instance();  37 : 46 : mDataProc = new VSBDataProcessor();  38 : 46 : mDataProc->initialize( pParams->getCellCommonData(),  39 : : pParams->getCellData(), pParams->getRefData(),  40 : : pParams->getLinkData() );  41 : : const DPMANAGER\_MESSAGE\_CONVERTER \*pStartMsg  42 : 46 : = pParams->getStartupMessage();  43 : 46 : uint64\_t blkSizeX = pStartMsg->ullVchipBSizeX;  44 : 46 : uint64\_t blkSizeY = pStartMsg->ullVchipBSizeY;  45 : 46 : size\_t numBlockX = pStartMsg->ullDpbWidth / blkSizeX;  46 : 46 : size\_t numBlockY = pStartMsg->ullDpbHeight / blkSizeY;  47 : 46 : uint32\_t numContainers = 0;  48 : : //TODO::Consider orientation while traversing blocks?  49 : 46 : boost::mutex::scoped\_lock l( mMutex );  50 : 46 : bool cellsFinished = false;  51 [ + - ][ # # ]: 46 : for( size\_t i = 0; i < numBlockX; i++ )  52 : : {  53 [ + - ][ # # ]: 46 : for( size\_t j = 0; j < numBlockY; j++ )  54 : : {  55 : 46 : mDataProc->setBlockIDs( i, j );  56 : 95 : while(true)  57 : : {  58 : 95 : Cell \*pCell = new Cell();  59 : 95 : bool cellFound=false;  60 : 95 : try{  61 : 95 : cellFound = mDataProc->getNextCell(pCell);   62 : : }  63 [ # # ]: 0 : catch(DPL::Error \*pChild)  64 : : {  65 : 0 : ErrorHandler \*pErr = ErrorHandler::instance();  66 : 0 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_,  67 : : \_\_LINE\_\_, ErrorHandler::eErrFunction,  68 : : pErr->getErrorMsg(  69 : : ErrorHandler::eErrFunction,  70 : : "ArbitraryModeProcessor::operator()"),  71 : : getThreadManager());  72 : : }  73 [ + + ]: 95 : if( cellFound )  74 : : {  75 : 51 : if( pCell->isArray() )  76 : : {  77 : 7 : SDDB \*pSDDB = SDDB::instance();  78 : 7 : try  79 : : {  80 : 7 : pSDDB->ArrayCellFound(\*pCell);  81 : : }  82 [ # # ]: 0 : catch(DPL::Error \*pChild)  83 : : {  84 : 0 : ErrorHandler \*pErr = ErrorHandler::instance();  85 : 0 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_,  86 : : \_\_LINE\_\_, ErrorHandler::eErrFunction,  87 : : pErr->getErrorMsg(  88 : : ErrorHandler::eErrFunction,  89 : : "SDDB::ArrayCellFound"),  90 : : getThreadManager());  91 : 0 : return;  92 : : }  93 : :   94 : : //Array Cell flattening  95 : 7 : IntPosition inBlockPos(i, j);  96 : 7 : IntPosition outBlockPos;  97 : :   98 : 7 : FlattenedContext flattenContext;  99 : 7 : flattenContext.setIsMoreCell(true);  100 : 7 : flattenContext.setNumCellX(0);  101 : 7 : flattenContext.setNumCellY(0);  102 : 28 : while(true)  103 : : {  104 : 28 : Cell \*outCell = new Cell();   105 [ + + ]: 28 : if(!mDataProc->getNextFlattenedCell(  106 : : \*pCell, outCell, inBlockPos, outBlockPos,  107 : : &flattenContext))  108 : : {  109 [ + - ]: 7 : delete outCell;  110 : 21 : break;  111 : : }  112 : 21 : try  113 : : {  114 : 21 : createContainer( outCell );  115 : : }  116 [ # # ]: 0 : catch(DPL::Error \*pChild)  117 : : {  118 : 0 : ErrorHandler \*pErr = ErrorHandler::instance();  119 : 0 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_,  120 : : \_\_LINE\_\_, ErrorHandler::eErrFunction,  121 : : pErr->getErrorMsg(   122 : : ErrorHandler::eErrFunction,  123 : : "SlidingWindow::scheduleSingleCell"),  124 : : getThreadManager());  125 : 0 : return;  126 : : }  127 : 21 : try  128 : : {  129 : 21 : pSDDB->SingleCellExtractedFromArrayCell(outCell);  130 : : }  131 [ # # ]: 0 : catch(DPL::Error \*pChild)  132 : : {  133 : 0 : ErrorHandler \*pErr = ErrorHandler::instance();  134 : 0 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_,  135 : : \_\_LINE\_\_, ErrorHandler::eErrFunction,  136 : : pErr->getErrorMsg(  137 : : ErrorHandler::eErrFunction,  138 : : "SDDB::SingleCellExtractedFromArrayCell"),  139 : : getThreadManager());  140 : 0 : return;  141 : : }  142 : :   143 : : }  144 : 7 : try  145 : : {  146 : 7 : pSDDB->ArrayCellFlatteningCompleted();  147 : : }  148 [ # # ]: 0 : catch(DPL::Error \*pChild)  149 : : {  150 : 0 : ErrorHandler \*pErr = ErrorHandler::instance();  151 : 0 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_,  152 : : \_\_LINE\_\_, ErrorHandler::eErrFunction,  153 : : pErr->getErrorMsg(  154 : : ErrorHandler::eErrFunction,  155 : : "SDDB::ArrayCellFlatteningCompleted"),  156 : : getThreadManager());  157 : 0 : return;  158 : : }  159 [ + - ]: 7 : delete pCell;  160 : : }  161 : : else //Cell is single cell  162 : : {  163 : 44 : try  164 : : {  165 : 44 : createContainer( pCell );  166 : : }  167 [ + - ]: 2 : catch(DPL::Error \*pChild)  168 : : {  169 : 2 : ErrorHandler \*pErr = ErrorHandler::instance();  170 : 2 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_,  171 : : \_\_LINE\_\_, ErrorHandler::eErrFunction,  172 : : pErr->getErrorMsg(   173 : : ErrorHandler::eErrFunction,  174 : : "SlidingWindow::scheduleSingleCell"),  175 : : getThreadManager());  176 : 2 : return;  177 : : }  178 : : }  179 : : }  180 : : else  181 : : {  182 [ + - ]: 44 : delete pCell;  183 [ + - ][ + - ]: 44 : if( i == numBlockX - 1 && j == numBlockY - 1 )  184 : : {  185 : 44 : e9k\_debug("All cells scheduled\n");  186 : 44 : cellsFinished = true;  187 : : }  188 [ + - ][ + - ]: 44 : if( numContainers < mMaxScheduledContainers  189 : : && !cellsFinished )  190 : : {  191 : 49 : break;  192 : : }  193 : : }  194 [ + - ]: 49 : if( !cellsFinished )  195 : : {  196 : 49 : numContainers++;  197 : : }  198 [ + - ][ + + ]: 93 : if( numContainers == mMaxScheduledContainers  199 : : || cellsFinished )  200 : : {  201 : 44 : numContainers = 0;  202 : 44 : int32\_t cellProcIdx = 0;  203 : 44 : for( ArbitraryModeCellProcessorList::iterator it  204 : 44 : = mCellProcessors.begin();  205 : : it != mCellProcessors.end(); it++ )  206 : : {  207 : 44 : ArbitrarySchedulingInfo cellProcInfo = \*it;  208 [ + - ]: 44 : if( cellProcInfo.second )  209 : : {  210 : : ArbitraryModeCellProcessor::Pointer cellProc  211 : 44 : = cellProcInfo.first;  212 : 44 : cellProc->notifyCellSchedulingCompletion();  213 : 44 : mNumActiveProcessors++;  214 : 44 : cellProcInfo.second = false;  215 : 44 : mCellProcessors[cellProcIdx] = cellProcInfo;  216 : 44 : e9k\_debug("Notifying processor %d (%p) of cell availability\n", cellProcIdx, cellProc);  217 : : }  218 : 44 : cellProcIdx++;  219 : : }  220 [ + - ][ + - ]: 44 : while( !mStop && mNumActiveProcessors != 0 )  221 : : {  222 : 44 : mWaitCondition.wait( l );  223 : : }  224 : 44 : e9k\_debug("All processors finished\n");  225 [ - + ]: 44 : if( mStop )  226 : : {  227 : 0 : return;  228 : : }  229 [ - + ]: 44 : if( cellsFinished )  230 : : {  231 : 44 : e9k\_debug("All cells finished.\n");  232 : 44 : mOutputGenerator->setIsComplete( true );  233 : 44 : return;  234 : : }  235 : : }  236 : : }  237 : : }  238 : : }  239 : 0 : int32\_t i = 0;  240 : : #if 0  241 : : for( ArbitraryModeCellProcessorList::iterator it  242 : : = mCellProcessors.begin(); it != mCellProcessors.end(); it++ )  243 : : {  244 : : ArbitrarySchedulingInfo cellProcInfo = \*it;  245 : : if( cellProcInfo.second )  246 : : {  247 : : ArbitraryModeCellProcessor::Pointer cellProc  248 : : = cellProcInfo.first;  249 : : cellProc->notifyCellSchedulingCompletion();  250 : : mNumActiveProcessors++;  251 : : cellProcInfo.second = false;  252 : : mCellProcessors[i] = cellProcInfo;  253 : : i++;  254 : : e9k\_debug("Notifying processor of cell availability\n");  255 : : }  256 : : }  257 : : while( !mStop && mNumActiveProcessors != 0 )  258 : : {  259 : : mWaitCondition.wait( l );  260 : : }  261 : : if( mStop )  262 : : {  263 : : return;  264 : : }  265 : : #endif  266 : : //Cell scheduling complete  267 : : }  268 : :   269 : : void  270 : 46 : ArbitraryModeProcessor::stop() {  271 : 46 : boost::mutex::scoped\_lock l(mMutex);  272 : 46 : mStop = true;  273 : 46 : mWaitCondition.notify\_one();  274 : : }  275 : :   276 : : void  277 : 44 : ArbitraryModeProcessor::notifyProcessingCompletion() {  278 : 44 : boost::mutex::scoped\_lock l(mMutex);  279 : 44 : mNumActiveProcessors--;  280 : 44 : mWaitCondition.notify\_one();  281 : : }  282 : :   283 : : void  284 : : ArbitraryModeProcessor::addCellProcessor(  285 : 46 : ArbitraryModeCellProcessor::Pointer inProcessor ) {  286 : 46 : mCellProcessors.push\_back( std::make\_pair( inProcessor, false ) );  287 : : }  288 : :   289 : : void  290 : 65 : ArbitraryModeProcessor::createContainer( Cell::Pointer inCell ) {  291 : 65 : try  292 : : {  293 : 65 : SDDB::instance()->cellSchedulingStart(inCell);  294 : : }  295 [ # # ]: 0 : catch(DPL::Error \*pChild)  296 : : {  297 : 0 : ErrorHandler \*pErr = ErrorHandler::instance();  298 : 0 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_,  299 : : ErrorHandler::eErrFunction,  300 : : pErr->getErrorMsg( ErrorHandler::eErrFunction,  301 : : "TFOrderEnforcer::performOrdering"));  302 : : }  303 : 65 : ConverterParams \*pParams = ConverterParams::instance();  304 : : IntRect cellArea( inCell->startPos,  305 : 65 : IntPosition( inCell->sc.dx, inCell->sc.dy ) );  306 : :   307 : 65 : SFTable::Pointer sfTable = NULL;  308 : 65 : try  309 : : {  310 : 65 : sfTable = SFTableCreator::constructSFTable( pParams, cellArea );  311 : : }  312 [ + - ]: 2 : catch( DPL::Error \*pChild )  313 : : {  314 : 2 : ErrorHandler \*pErr = ErrorHandler::instance();  315 : 2 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_,  316 : : ErrorHandler::eErrFunction,  317 : : pErr->getErrorMsg( ErrorHandler::eErrFunction,  318 : : "SFTableCreator::constructSFTable"));  319 : : }  320 : : ArbitraryModeCellContainer::Pointer pCellContainer  321 : 63 : = new(std::nothrow) ArbitraryModeCellContainer();  322 : : //#ifdef ERROR\_SEED //<<Test\_Id:pCellContainer\_Set\_To\_false>>  323 : : // pCellContainer = false;  324 : : //#endif  325 [ - + ]: 63 : if( !pCellContainer )  326 : : {  327 : 0 : int iErrNo = errno; //System error no  328 : 0 : char systemErrorMsg[BUFSIZ];  329 : 0 : ErrorHandler \*pErr = ErrorHandler::instance();  330 : 0 : pErr->errorAdd(NULL, \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_,  331 : : ErrorHandler::eErrMemoryAllocation,  332 : : pErr->getErrorMsg(  333 : : ErrorHandler::eErrMemoryAllocation,  334 : : strerror\_r( iErrNo, systemErrorMsg, BUFSIZ )));  335 : : }  336 : 63 : pCellContainer->setTargetCell( inCell );  337 : 63 : pCellContainer->setSFT( sfTable );  338 : 63 : pCellContainer->setCellPosition( inCell->startPos );  339 : 63 : mOutputGenerator->addCellForOutput( pCellContainer );  340 : : //Schedule Cell  341 : 63 : size\_t numThreads = pParams->getStartupMessage()->usConcurrency;  342 [ + + ]: 63 : if( 0 == mThreadDataSizes )  343 : : {  344 : 44 : try  345 : : {  346 : 44 : mThreadDataSizes = (size\_t \*) nftMalloc( sizeof(size\_t)  347 : : \* numThreads );  348 : : }  349 [ # # ]: 0 : catch(DPL::Error \*pChild)  350 : : {  351 : 0 : ErrorHandler \*pErr = ErrorHandler::instance();  352 : 0 : pErr->errorAdd(pChild, \_\_FILE\_\_, \_\_FUNCTION\_\_, \_\_LINE\_\_,  353 : : ErrorHandler::eErrFunction,  354 : : pErr->getErrorMsg( ErrorHandler::eErrFunction,  355 : : "nftMalloc"));  356 : : }  357 : 44 : bzero( mThreadDataSizes, sizeof(size\_t) \* numThreads);  358 : : }  359 : 63 : size\_t minDataId = 0;  360 : 63 : size\_t minDataSize = std::numeric\_limits<size\_t>::max();  361 : 63 : size\_t dataSize = inCell->getSegmentDataSize();  362 [ + - ][ - + ]: 126 : for( size\_t i = 0; i < numThreads; i ++ )  363 : : {  364 : 63 : size\_t \*pDataSize = mThreadDataSizes + i;  365 [ + - ]: 63 : if( \*pDataSize + dataSize < minDataSize )  366 : : {  367 : 63 : minDataSize = \*pDataSize + dataSize;  368 : 63 : minDataId = i;  369 : : }  370 : : }  371 : 63 : size\_t \*pDataSize = mThreadDataSizes + minDataId;  372 : 63 : \*pDataSize = minDataSize;  373 : 63 : (mCellProcessors[ minDataId ].first)->scheduleCell( pCellContainer );  374 : 0 : mCellProcessors[ minDataId ].second = true;  375 : 63 : e9k\_debug("Cell at %d,%d was scheduled to thread %llu\n",  376 : : inCell->startPos.getX(), inCell->startPos.getY(), minDataId);  377 : : }  378 : :   379 : 1215 : } // namespace nft |

|  |
| --- |
| Generated by: [LCOV version 1.9](http://ltp.sourceforge.net/coverage/lcov.php) |